

AFRL-IF-RS-TR-2002-296
Final Technical Report
November 2002



MALLEABLE CACHES

Massachusetts Institute of Technology

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J203


APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.


The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-296 has been reviewed and is approved for publication.

APPROVED: 
RAYMOND A. LIUZZI
Project Engineer


FOR THE DIRECTOR:
MICHAEL L. TALBERT, Maj., USAF
Technical Advisor, Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE NOVEMBER 2002	3. REPORT TYPE AND DATES COVERED Final May 99 – Sep 01	
4. TITLE AND SUBTITLE MALLEABLE CACHES			5. FUNDING NUMBERS C - F30602-99-2-0511 PE - 62301E PR - HPSW TA - 00 WU - 06	
6. AUTHOR(S) Srini Devadas and Larry Rudolph				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Massachusetts Institute of Technology 545 Technology Square Cambridge Massachusetts 02139			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Project Agency AFRL/ITFB 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2002-296	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Raymond A. Liuzzi/ITFB/(315) 330-3577/ Raymond.Liuzzi@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) Managing the memory hierarchy is important for providing good performance of data intensive computation. This effort has explored several techniques for managing the cache in a microprocessor. This report examines column caching, cache partitioning, and cache compression techniques, especially in regards to the Data Intensive System (DIS) benchmarks. As a result of this study it was found that compression can be added to caches to improve capacity, but creates problems of replacement strategy and fragmentation. These problems can be solved using partitioning. A dictionary-based compression scheme allows for reasonable compression and decompression latencies and compression ratios. Keeping the data in the dictionary from becoming stale can be avoided with a clock scheme. The performance gains of a PCC over a standard cache of equivalent size can be attributed to two factors. A PCC potentially stores more data than a standard cache, which can reduce capacity misses and a PCC has more associativity than a standard cache of equivalent size, which can reduce conflict misses. Various techniques can be used to reduce the latency involved in the compression and decompression process. Searching on part of the dictionary during compression, using multiple banks or CAMs to examine multiple dictionary entries simultaneously, and compressing a cache line starting at different points in parallel can reduce compression latency. Finally, there are many different compression schemes some of which may perform better or be easier to implement in hardware.				
14. SUBJECT TERMS Computer Architecture, Storage, Data Intensive Computing, Hardware/Software, Computer Memory				15. NUMBER OF PAGES 57
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table Of Contents

1. Introduction.....	1
2. Cache Partitioning.....	1
2.1 Example 1 Stressmarks	3
2.2 Marginal-Gain Counters	3
2.3 Memory-Aware Scheduling.....	8
2.4 Cache Partitioning.....	12
2.5 Analytical Models.....	15
2.6 Example 2-Dynamic Partitioning	17
2.7 Experimental Verification.....	25
3. Cache Partitioning.....	28
3.1 Recording Memory Reference Patterns.....	28
3.2 The Partitioning Scheme.....	30
4. Compressed Caches	33
4.1 Related Word	34
4.2 The PCC Compression Algorithm.....	35
4.3 PCC Management	38
4.4 The Performance of PCC	41
4.5 Conclusions.....	46
References.....	46

List of Figures

Figure 1: (a) Miss-rate curve for process A (gcc). (b) Miss-rate curve for process B (swim). (c) Miss-rate curve for process C (bzip2).	2
Figure 2: The implementation of memory monitors for main memory	5
Figure 3: (a) The implementation that only uses the LRU information within a set. (b) The implementation that uses both the way LRU information and the set LRU information.....	6
Figure 4: (a) Approximation only using the way LRU information. (b) Approximation using both the way LRU information and the set LRU Information.....	8
Figure 5: (a) A shared memory multiprocessor system with P processors. (b) Space-sharing and Time-sharing in multiprocessor system	9

Figure 6:	The comparison of miss-rates for various schedules: the worst case, the best case, and the schedule decided by the algorithm.	11
Figure 7:	The implementation of on-line cache partitioning.....	13
Figure 8:	The comparison of miss-rates for various schedules: the worst case, the best case, the scheduled based on the model, and the schedule decided by the algorithm in Section 2.3.....	16
Figure 9:	The results of cache partitioning among concurrent processes	17
Figure 10:	(a) The overview of an analytical cache model. (b) Round-Robin Schedule	17
Figure 11:	(a) The probability of a miss at time t_0 . (b) The number of misses from $P_{miss}(t)$ curve.....	18
Figure 12:	The snapshot of a cache after running Process i for time t	21
Figure 13:	The relation between $x_i^\phi(t)$ and $x_i(t)$. $x_i(0)$ is the amount of Process i 's data in the cache when a time quantum starts.....	23
Figure 14:	The result of a cache model for cache flushing cases. (a) vpr. (b) vortex. (c) gcc. (d) bzip2.....	24
Figure 15:	The result of the cache model when two processes (vpr, vortex) are sharing a cache (32 KB fully-associative). (a) the overall miss-rate. (b) the initial amount of data $x_i(0)$	26
Figure 16:	The overall miss-rate when four processes (vpr, vortex, gcc, bzip@) are sharing a cache (32 KB, fully-associative).....	27
Figure 17:	The implementation of on-line cache partitioning.....	28
Figure 18:	The characteristics of the benchmarks. (a) The change of a miss-rate over time. (b) The miss-rate as a function of the cache size	29
Figure 19:	(a) The average miss-rate for various time quanta. (b) the change of the miss-rate over time with ten memory references per time quantum	32
Figure 20:	The results of the model-based cache partitioning for a set-associative cache when eight processes (bzip2, gcc, swim, mesa, vortex, vpr, twolf, iu) are sharing the cache (32 KB, 8-way associative).....	33
Figure 21:	The space-efficient dictionary stores only one uncompressed symbol per entry, while the reduced-latency dictionary stores the entire string.	36
Figure 22:	Compression and Decompression Logic	37
Figure 23:	The histograms indicate the amount of data available at different levels of compressibility.....	39
Figure 24:	A sample configuration in which cache lines are assumed to be compressible to $\frac{1}{2}$ their size, e.g. a 32 byte line compressed requires only 16 bytes	40
Figure 25:	To the left of the knee, small increases in IMREC raio correspond to large increases in MRR.	43
Figure 26:	We plot the art and equake IMREC raio over time so as to know how long to simulate before recording results.....	43
Figure 27:	The IMREC values for five benchmarks show that PCC clearly improves the performance	44
Figure 28:	The ATQ values for five benchmarks show that even when the latency of decompressing cache items, PCC still can improve the performance although not for all applications.....	45
Figure 29:	The IMREC values for the modified PCC replacement strategy that tries to keep the MRU item decompressed.....	45
Figure 30:	The ATQ values corresponding to Figure 29	45

List of Tables

Table 1:	The descriptions and Footprints of benchmarks used for the simulations.....	10
Table 2:	The performance of the memory-aware scheduling algorithm	11
Table 3:	The benchmark sets simulated	14
Table 4:	Hit-rate Comparison between the standard LRU and the partitioned LRU.....	14
Table 5:	Replacement Algorithm	38

Malleable Caches: Final Report

Srini Devadas and Larry Rudolph
Laboratory for Computer Science
MIT
Cambridge, MA 02139

1 Introduction

Managing the memory hierarchy is important for providing good performance of data intensive computation. We have explored several techniques for managing the cache in a microprocessor. In this report, we look at column caching, cache partitioning, and cache compression techniques especially in regards to the DIS benchmarks.

2 Cache Partitioning

We present a low-overhead, on-line memory monitoring scheme that is more useful than simple cache hit counters. The scheme becomes increasingly important as more and more processes and threads share various memory resources in computers using SMP [28, 33, 34], Multiprocessor-on-a-chip [29], or SMT [47, 36, 30] architectures.

Regardless of whether a single process executes on the machine at a given point in time, or multiple processes execute simultaneously, modern systems are *space-shared* and *time-shared*. Since multiple processes or threads¹ can interfere in memory or caches, the performance of a process can depend on the actions of other processes. Despite the importance of optimizing memory performance for multi-tasking situations, most published research focuses only on improving the performance of a single process.

Optimizing memory usage between multiple processes is virtually impossible without run-time information. The processes that share resources in the memory hierarchy are only known at run-time, and the memory reference characteristic of each process, heavily depends on inputs to the process and the phase of execution. But, hardware cache monitors in commercial, general-purpose microprocessors (e.g., [48]) only count the total number of misses which is useful for pure performance monitoring of a single application.

¹We use the term “process” in the paper to potentially include any execution context, such as threads. Too bad there is no consistent use of these terms.

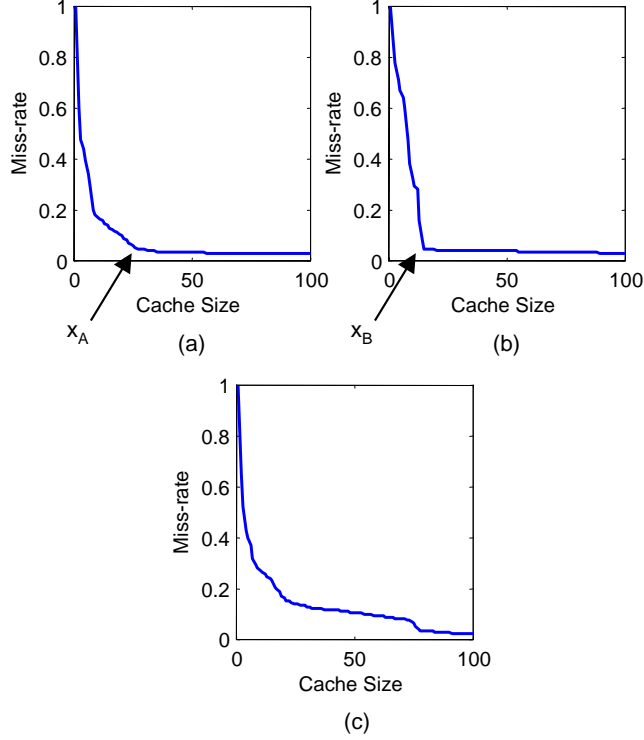


Figure 1: (a) Miss-rate curve for process *A* (*gcc*). (b) Miss-rate curve for process *B* (*swim*). (c) Miss-rate curve for process *C* (*bzip2*).

To determine how many and which jobs should execute simultaneously, it is often necessary to know how an application would perform for various cache sizes. Cache “footprint” for each application usually does not help since footprints for several applications executing simultaneously are likely to exceed the cache size for small caches. For example, consider the miss-rate curves for three different processes from SPEC CPU2000 [32] shown in Figure 1. For a cache of size 50, *A* and *B* could execute together but *C* should execute alone. Miss-rates as a function of cache size give much more information than a single footprint number and this information can be very relevant in scheduling, and partitioning cache among processes.

The memory monitoring scheme presented in this paper requires small modifications to the TLB, L1, and L2 cache controllers and the addition of a set of counters. Despite the simplicity of the hardware, these counters provide isolated miss-rates of each running process as a function of cache size under the standard LRU replacement policy². Moreover, the monitoring information can be used to dynamically reflect changes in process’ behavior by properly weighting counters’ values.

In our scheduling and partitioning algorithms (Section 2.3, 3), we use marginal gains rather than miss-rate curves. The marginal gain of process *i*, namely $g_i(x)$, is defined as the derivative of

²Previous approaches only produce a single number corresponding to one memory size.

the miss-rate curve ³ $m_i(x)$ properly weighted by the number of references for a processes (ref_i);

$$g_i(k) = (m_i(k-1) - m_i(k)) \cdot ref_i. \quad (1)$$

Therefore, we directly monitor marginal gains for each process rather than miss-rate curves. Using marginal gains, we can derive schedules and cache allocations for jobs to improve memory performance. If needed, miss-rate curves can be computed recursively from marginal gains.

We show how the information from the memory monitors is analyzed using an analytical framework, which models the effects of memory interference amongst simultaneously-executing processes as well as time-sharing effects (Section 2.5). The counter values alone only estimate the effects of reducing cache space for each process. When used in conjunction with the analytical model, they can provide an accurate estimate of the overall miss-rate of a set of processes time-sharing and space-sharing a cache. The overall miss-rate provided by the model can drive more powerful scheduling and partitioning algorithms.

2.1 Example 1 – Stressmarks

2.2 Marginal-Gain Counters

Memory monitoring schemes should provide information to estimate the performance of a given level of the memory hierarchy under different configurations or allocations to be useful when optimizing that level’s performance. This section proposes an architectural mechanism using a set of counters to obtain the *marginal-gain* in cache hits for different sizes of the cache for a process or set of processes. Such information is used by memory-aware scheduling and partitioning schemes.

For fully-associative caches, the counters simply indicate the marginal gains, but for set-associative caches, the counters are mapped to marginal gains for an equivalent sized fully-associative cache. It is much easier to work with fully-associative caches and experimental results show that this works well in practice. For example, the contention between two processes sharing a fully-associative cache is a good approximation to the contention between the two processes sharing a set-associative cache.

2.2.1 Implementation of Counters

We want to obtain marginal gains for a process for various cache sizes without actually changing the cache configuration. In cache simulations, it has been shown that different cache sizes can be simulated in a single pass [41]. We emulate this technique in hardware to obtain multiple marginal gains while executing a process with a fixed cache configuration.

In any situation where the exact LRU ordering of each cache block is known, computing the marginal gain $g(x)$ simply follows from the following set of counters:

³the miss-rate of process i using x cache blocks when the process is isolated without competing processes.

Counters for a Fully Associative Cache: There is one counter for each block in the cache; $counter(1)$ records the number of hits in the most recently used block, and $counter(2)$ is the number of hits in the second most recently used block, etc. When there is a reference to the i^{th} most recently used block, then $counter(i)$ is incremented. Note that the item referenced then becomes the most recently used block, so that a subsequent reference to that item is likely to increment a different counter.

To compute the marginal gain curve for each process, a set of counters is maintained for each process. In a uniprocessor system, the counters are saved/restored during context switches, and when processes execute in parallel, multiple sets of counters are maintained in hardware. We thus subscript the counters with their associated process id. The marginal gain $g_i(x)$ is obtained directly by counting the number of hits in the x^{th} *most recently used* block ($counter(x)$). The counters plus an additional one, ref_i , that records the total number of cache references for process i , are used to convert marginal gains to miss-rates for analytical models (Section 2.5).

2.2.2 Main Memory

Main memory can be viewed as a fully-associative cache for which on-line marginal gain counters could be useful. That is, we want to know the marginal gain to a process as a function of physical memory size. For main memory, there are two different types of accesses that must be considered: a TLB hit or a TLB miss. Collecting marginal gain information from activity associated with a TLB hit is important for processes that have small footprints and requires hardware counters in the TLB. Collecting this information when there is a TLB miss is important for processes with larger footprints and requires mostly software support.

Assuming the TLB is a fully-associative cache with LRU replacement, the hardware counters defined above can be used to compute marginal gains for the C_{TLB} most recently used pages, where C_{TLB} is the number of TLB entries, Figure 2. The counters are only increased if a memory access misses on both L1 and L2 caches. Therefore, counting accesses to main memory does not introduce additional delay on any critical path. If the TLB is set-associative we use the technique described in the next subsection.

On a TLB miss, a memory access is serviced by either a hardware or software TLB miss handler. Ideally, we want to maintain the LRU ordering for each page and count hits per page. However, the overhead of per-page counting is too high and experimentation shows that only dozens of data points are needed for performance optimization such as scheduling and partitioning. Therefore, the entire physical memory space can be divided into a few dozen groups and we count the

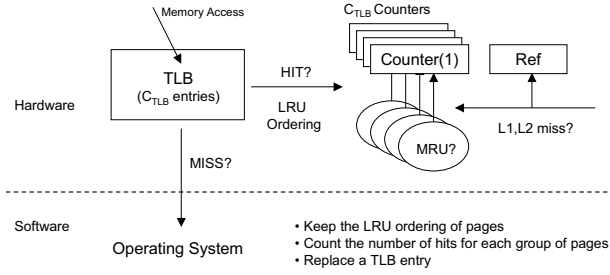


Figure 2: The implementation of memory monitors for main memory.

marginal gain per group. It is easy for software to maintain the LRU information. All of a process' pages in physical memory form a linked list in LRU ordering. When the page is accessed, its group counter is updated, its position on the linked list is moved to the front, and all the pages on group boundaries update their group. Machines that handle TLB misses in hardware need only insert the referenced page number into a buffer and software can do the necessary updates to the linked list on subsequent context switches. The overhead is minor requiring only several bytes for each page whose size is of the order of 4-KB, and tens of counters to compute marginal gains.

2.2.3 Set-Associative Caches

In set-associative caches, LRU ordering is kept only within each set. (We call this LRU ordering within a set as *way LRU ordering*.) Although we can only estimate marginal gains of having each *way*, not each cache block, it turns out to often to be good enough for scheduling and partitioning if the cache has reasonably high associativity.

Way-Counters for a Set-Associative Cache: There is one counter for each way of the cache. A hit in the cache to the MRU block of some set updates $counter(1)$. A hit in the cache to the LRU block of some set updates $counter(D)$, assuming D -way associativity. There is an additional counter, ref , recording all the accesses to the cache.

Figure 3 (a) illustrates the implementation of this hardware counters for 2-way associative caches. It is also possible to have counters associated with each set of a cache.

Set-Counters for a Set-Associative Cache: There is one counter for each set of the cache. LRU information for all sets is maintained. A hit to any block within the MRU set updates $counter(1)$. A hit to any block within the LRU set updates $counter(S)$, assuming S sets in the cache. There is an additional counter, ref , recording all the accesses to the cache.

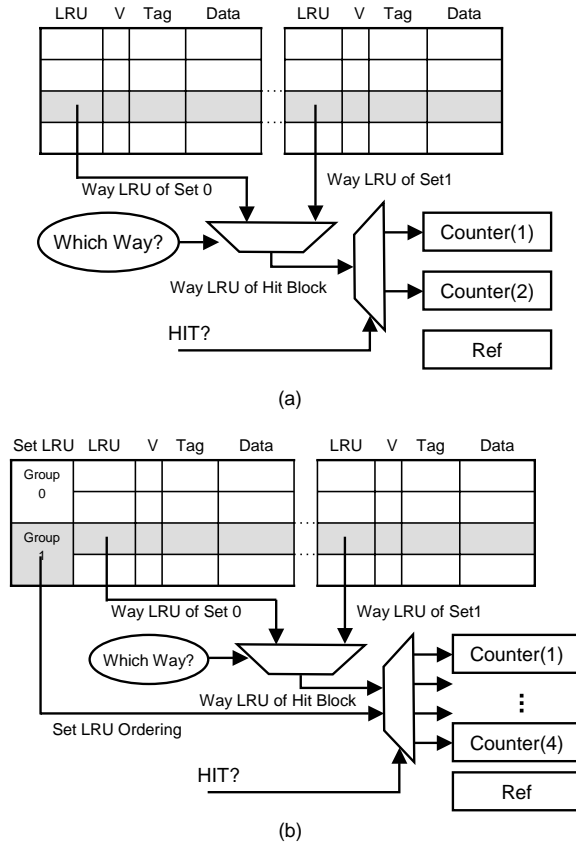


Figure 3: The implementation of memory monitors for 2-way associative caches. On a cache access, the LRU information is read for the accessed set. Then the counter is incremented based on this LRU information if the access hits on the cache. The reference counter is increased on every access. (a) The implementation that only uses the LRU information within a set. (b) The implementation that uses both the way LRU information and the set LRU information.

To obtain the most detailed information, we can combine both *way-counters* and *set-counters*.

There are $D \cdot S$ counters, one for each cache block. A hit to a block within the i^{th} MRU set and the j^{th} MRU way updates $counter(i, j)$. We refer to these as *DS-counters*.

In practice, we do not need to maintain LRU ordering on a per cache set basis. Since there could be thousands of cache sets, the sets are divided into several groups and the LRU ordering is maintained for the groups. Figure 3 (b) illustrates the implementation of DS-counters with two set groups.

2.2.4 Computing fully-associative marginal gain from set-associative counters

The marginal gain for a fully-associative cache can be approximated from the way-counters as follows:

$$counter_i(k) = \sum_{x=(k-1) \cdot S+1}^{k \cdot S} g_i(x) \quad (2)$$

where S is the number of sets.

With a minimum monitoring granularity of a *way*, high-associativity is essential for obtaining enough information for performance optimization; our experiments show that 8-way associative caches can provide enough information for partitioning. Content-addressable-memory (CAM) tags are attractive for low-power processors [49] and they have higher associativity; the SA-1100 StrongARM processor [35] contains a 32-way associative cache.

If the cache has low associativity, the information from the way LRU ordering alone is often not enough for good performance optimization. For example, consider a 2-way associative cache shown in Figure 4 (a). For cache partitioning, the algorithm would conclude that the process needs a half of the cache to achieve a low miss-rate from two given points, even though the process only needs one tenth of the cache space.

To obtain finer-grained information, we use either *Way-Counter* with *Set-Counters* or *DS-Counters* for low-associative caches. For example, Figure 4 (b) shows the miss-rate curve obtained using DS-Counters. As shown in the figure, we can obtain much more detailed information if we keep the set LRU ordering for 8 or 16 groups. Way-Counters with Set-Counters, which provide $D + S$ counter values, can also be used instead of DS-Counters. In this case, the value in each set-counter is distributed over the ways (D software counters) based on the values in the way-counters to generate $D \cdot S$ values.

There are several strategies for converting the $D \cdot S$ counter values into full-associative marginal gain information. In Figure 4 (b), we used *sorting* as a conversion method. First, $D \cdot S_{group}$ counter values are obtained from the hardware counters, where S_{group} represents the number of set groups. Then, these counters are sorted in decreasing order and assigned to marginal gains. This conversion is based on the assumption that the marginal gain is monotonically decreasing function of cache size. We are also investigating other conversion methods; column-major conversion, binomial probability conversion, etc.

Since characteristics of processes change dynamically, the estimation of $g_i(x)$ should reflect the changes. But we also wish to maintain some history of the memory reference characteristics of a process, so we can use it to make decisions. We can achieve both objectives, by giving more weight to the counter value measured in more recent time periods.

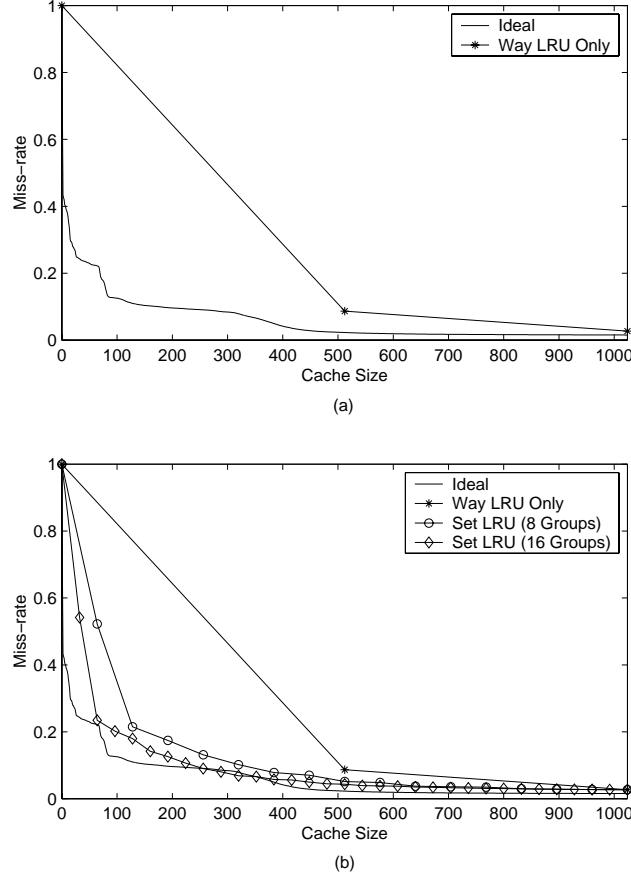


Figure 4: The estimated miss-rate curves using the set-associative cache monitor. The cache is 32-KB 2-way associative, and the benchmark is `vpr` from SPEC CPU2000. The ideal curve represents the case when you know the LRU ordering of all cache blocks. (a) Approximation only using the way LRU information. (b) Approximation using both the way LRU information and the set LRU information.

When a process begins running for the first time, all its counter values are initialized to zero. At the beginning of each time quantum that process i runs, the operating system multiplies $counter_i(k)$ for all k and ref_i by $\delta = 0.5$. As a result, the effect of hits in the previous time slice exponentially decays, but we maintain some history.

2.3 Memory-Aware Scheduling

When a scheduler has the freedom to select which processes execute in parallel, knowing the memory requirements of each process can help produce a better schedule. In particular, this section demonstrates how the marginal gain counters can be used to produce a memory-aware schedule. First, we begin with the problem definition and assumptions. Then, a scheduling algorithm based on marginal gains of each process is briefly explained. Finally, we validate our approach by simulations for main memory.

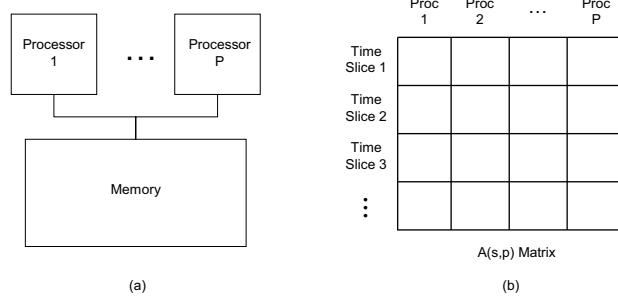


Figure 5: (a) A shared memory multiprocessor system with P processors. (b) Space-sharing and Time-sharing in multiprocessor system.

2.3.1 Scheduling Problem

We consider a system where P identical processors share the memory and N processes are ready to execute, see Figure 5 (a). The system can be a shared-memory multiprocessor system where multiple processors share the main memory, or it can be a chip multiprocessor system where processors on a single chip share the L2 cache.

Since there are P processors, a maximum of P processes can execute at the same time. To schedule more than P processes, the system is time-shared. We will assume processes are single-threaded, and all P processors context switch at the same time as would be done in gang scheduling [31]. These assumptions are not central to our approach, rather for the sake of brevity, we have focused on a basic scheduling scenario. There may or may not be constraints in scheduling the ready processes. Constraints will merely affect the search for feasible schedules.

A schedule is a mapping of processes to matrix elements, where element $A(s, p)$ represents the process scheduled on processor p for time slice s , see Figure 5 (b). A matrix with S non-empty rows indicates that S time slices are needed to schedule all N processes. In our problem, $S = \lceil \frac{N}{P} \rceil$.

Our problem is to find the optimal scheduling that minimizes processor idle time due to memory misses. The number of memory misses depends on both contention amongst processes in the same time slice and contention amongst different time slices. In this section, we only consider the contention within the time slice. Considering contention amongst time slices is briefly discussed in Section 2.5. For a more general memory-aware scheduling strategy, see [44].

2.3.2 Scheduling Algorithm

For many applications, the miss rate curve as a function of memory size has a knee (See Figure 1). That is, the miss rate quickly drops and then levels off. To minimize the number of misses, we want to schedule processes so that each process can use more cache space than the ordinate of its knee.

The relative footprint for process i is defined as the number of memory blocks allocated to the process when the memory with $C \cdot S$ blocks is partitioned among all processes such that the marginal gain for all processes is the same. C represents the number of blocks in the memory, and $C \cdot S$ represents the amount of available memory in S time slices. Effectively, the relative footprint of a process represents the optimal amount of memory space for that process when all processes

Name	Description	FP (MB)
bzip2	Compression	6.2
gcc	C Compiler	22.3
gzip	Compression	76.2
mcf	Combinatorial Optimization	9.9
vortex	Object-oriented Database	83.0
vpr	FPGA Placement and Routing	1.6

Table 1: The descriptions and Footprints of benchmarks used for the simulations. All benchmarks are from SPEC CPU2000 [32].

execute simultaneously sharing the total memory resource over S time slices ⁴. Intuitively, relative footprints corresponds to a knee of the miss-rate curve for a process.

We use a simple $C \cdot S$ step greedy algorithm to compute relative footprints. First, no memory block is allocated to any process. Then, for each block, we allocate the block to the process that obtains the maximum marginal gain for an additional block. After allocating all $C \cdot S$ blocks to processes, the allocation for each process is the relative footprint of the process. We limit the number of blocks assigned to each process to be less than or equal to C .

Once the relative footprints are computed, assigning processes to time slices is straightforward. In a greedy manner, the unscheduled process with the largest relative footprint is assigned to a time slice with the smallest total relative footprint at the time. We limit the number of processes for each time slice to be P .

2.3.3 Experimental Results

A trace-driven simulator demonstrates the importance of memory-aware scheduling and the effectiveness of our memory monitoring scheme. Consider scheduling six processes, randomly selected from SPEC CPU2000 benchmark suite [32] on the system with three processors sharing the main memory. The benchmark processes have various footprint sizes (See Table 1), that is, the memory size that a benchmark requires to achieve the minimum miss-rate. Processors are assumed to have 4-way 16-KB L1 instruction and data caches and a 8-way 256-KB L2 cache. The simulations concentrate on the main memory varying over a range of 8 MB to 256 MB with 4-KB pages.

All possible schedules are simulated. For various memory sizes, we compare the average miss-rate of all possible schedules with the miss-rates of the worst schedule, the best schedule, and the schedule by our algorithm. The simulation results are summarized in Table 2 and Figure 6. In the table, a corresponding schedule for each case is also shown except for the 128-MB and 256-MB cases where many schedules result in the same miss-rate. A schedule is represented by two sets of letters. Each set represents a time slice, and each letter represents a process: A-bzip2, B-gcc, C-gzip, D-mcf, E-vortex, F-vpr. In the figure, the miss-rates are normalized to the average miss-rate.

The results demonstrate that process scheduling can have a significant effect on the memory

⁴Stone, Turek, and Wolf [40] proved the algorithm results in the optimal partition assuming that marginal gains monotonically decrease as allocated memory increases.

Memory Size (MB)		Average of All Cases	Worst Case	Best Case	Algorithm
8	Miss-Rate(%)	1.379	2.506	1.019	1.022
	Schedule		(ADE,BCF)	(ACD,BEF)	(ACE,BDF)
16	Miss-Rate(%)	0.471	0.701	0.333	0.347
	Schedule		(ADE,BCF)	(ADF,BCE)	(ACD,BEF)
32	Miss-Rate(%)	0.187	0.245	0.148	0.157
	Schedule		(ADE,BCF)	(ACD,BEF)	(ABD,CEF)
64	Miss-Rate(%)	0.072	0.085	0.063	0.066
	Schedule		(ABF,CDE)	(ACD,BEF)	(ACF,BDE)
128	Miss-Rate(%)	0.037	0.052	0.029	0.029
	Schedule		(ABF,CDE)	(ACD,BEF)	(ACD,BEF)
256	Miss-Rate(%)	0.030	0.032	0.029	0.029
	Schedule		(ABF,CDE)	(ACD,BEF)	(ACD,BEF)

Table 2: The performance of the memory-aware scheduling algorithm. A schedule is represented by two sets of letters. Each set represents a time slice, and each letter represents a process: A-bzip2, B-gcc, C-gzip, D-mcf, E-vortex, F-vpr. For some cases multiple schedules result in the same miss-rate.

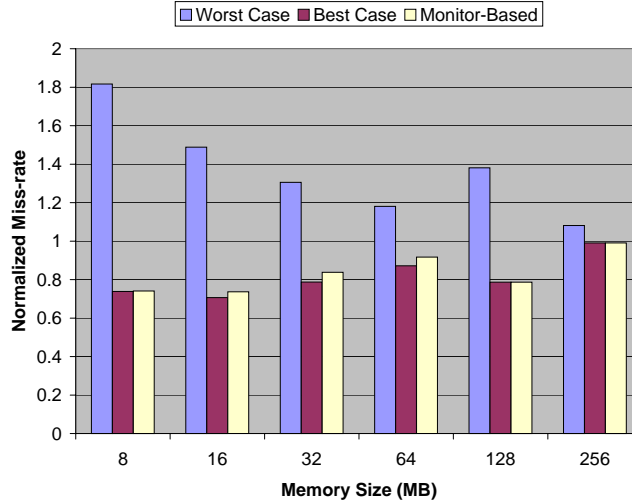


Figure 6: The comparison of miss-rates for various schedules: the worst case, the best case, and the schedule decided by the algorithm. The miss-rates are normalized to the average miss-rate of all possible schedules for each memory size.

performance, and thus the overall system performance. For 16-MB memory, the best case miss-rate is about 30% better than the average case, and about 53% better than the worst case. Given the very large penalty for a page fault, performance is significantly improved due to this large reduction in miss-rate. As the memory size increases, scheduling becomes less important since the entire workload fits into the memory. However, note that smart scheduling can still improve the miss-rate by about 10% over the worst case even for 256-MB memory that is larger than the total footprint size from Table 1. This happens because the LRU policy does not allocate the memory properly.

The results also illustrate that our scheduling algorithm can effectively find a good schedule, which results in a low miss-rate. In fact, the algorithm found the optimal schedule when the memory is larger than 64-MB. Even for small memory, the schedule found by the algorithm shows a miss-rate very close to the optimal case.

Finally, the results demonstrate the advantage of having marginal gain information for each process rather than one value of footprint size. If we schedule processes based on the footprint size, executing `gcc`, `gzip` and `vpr` together and the others in the next time slice seems to be natural since it balances the total footprint size for each time slice. However, this schedule is actually the *worst* schedule for memory smaller than 128-MB, and results in a miss-rate that is over 50% worse than the optimal schedule.

Memory traces used in this experiment have footprints smaller than 100 MB. As a result, the scheduling algorithm could not improve the miss-rate for memory which is larger than 256 MB. However, many applications have very large footprints, often larger than main memory. For these applications, the memory size where scheduling matters should scale up.

2.4 Cache Partitioning

Just like knowing memory requirements can help a scheduler, it can also be used to decide the best way to dynamically partition the cache among simultaneous processes. A partitioned cache explicitly allocates cache space to particular processes. In a partitioned cache, if space is allocated to one process, it cannot be used to satisfy cache misses by other processes. Using trace-driven simulations, we compare partitioning with normal LRU for set-associative caches.

2.4.1 The Partitioning Scheme

The standard LRU replacement policy treats all cache blocks in the same way. For multi-tasking situations, this can often result in poor allocation of cache space among processes. When multiple processes run simultaneously and share the cache as in simultaneous multithreading and chip multiprocessor systems, the LRU policy blindly allocates more cache space to processes that generate more misses even though other processes may benefit more from increased cache space.

We solve this problem by explicitly allocating cache space to each process. The standard LRU policy still manages cache space within a process, but not among processes. Each process gets a certain amount of cache space allocated explicitly. Then, the replacement unit decides which block within a set will be evicted based on how many blocks a process has in the cache and how many blocks are allocated to the process.

The overall flow of the partitioning scheme can be viewed as a set of four modules: on-line

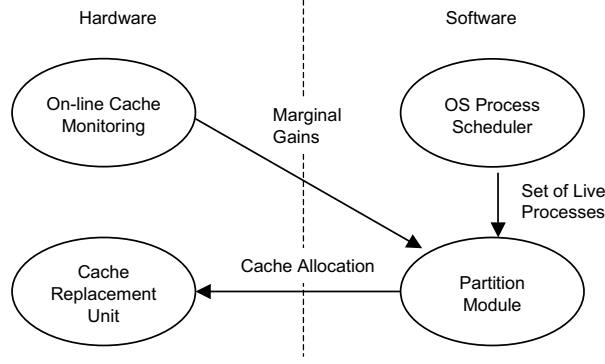


Figure 7: The implementation of on-line cache partitioning.

cache monitor, OS processor scheduler, partition module, and cache replacement unit (Figure 17). The scheduler provides the partition module with the set of executing processes that shares the cache at the same time. Then, the partition module uses this scheduling information and the marginal gain information from the on-line cache monitor to decide a cache partition; the module uses a greedy algorithm to allocate each cache block to a process that obtains the maximum marginal gain by having one additional block. Finally, the replacement unit maps these partitions to the appropriate parts of the cache. Since the characteristics of processes change dynamically, the partition is re-evaluated after every time slice. For details on the partitioning algorithm, see [43].

2.4.2 Experimental Results

This section presents quantitative results using our cache allocation scheme. The simulations concentrate on chip multiprocessor systems where processors (either 2 or 4) share the same L2 cache. The shared L2 cache is 8-way set-associative, whose size varies over a range of 256 KB to 4 MB. Each processor is assumed to have its own L1 instruction and data caches, which are 4-way 16 KB. Due to large space and long latency to main memory, our scheme is more likely to be useful for an L2 cache, and so that is the focus of our simulations. We note in passing, that we believe our approach will work on L1 caches as well if L1 caches are also shared.

Three different sets of benchmarks are simulated, see Table 3. The first set (Mix-1) has two processes, `art` and `mcf` both from SPEC CPU2000. The second set (Mix-2) has three processes, `vpr`, `bzip2` and `iu`. Finally, the third set (Mix-3) has four processes, two copies of `art` and two copies of `mcf`, each with a different phase of the benchmark.

The simulations compare the overall L2 miss-rate of a standard LRU replacement policy and the overall L2 miss-rate of a cache managed by our partitioning algorithm. The partition is updated every two hundred thousand memory references ($T = 200000$), and the counters are multiplied by $\delta = 0.5$ (cf. Section 2.2.4). Carefully selecting values of T and δ is likely to give better results. The hit-rates are averaged over fifty million memory references and shown for various cache sizes (see Table 4).

The simulation results show that the partitioning can improve the L2 cache miss-rate significantly: for cache sizes between 1 MB to 2 MB, partitioning improved the miss-rate up to 14%

Name	Process	Description
Mix-1	art	Image Recognition/Neural Network
	mcf	Combinatorial Optimization
Mix-2	vpr	FPGA Circuit Placement and Routing
	bzip2	Compression
	iu	Image Understanding
Mix-3	art1	Image Recognition/Neural Network
	art2	
	mcf1	Combinatorial Optimization
	mcf2	

Table 3: The benchmark sets simulated. All but the Image Understanding benchmark are from SPEC CPU2000 [32]. The Image Understanding is from DIS benchmark suite [38].

Size (MB)	L1 %Miss	L2 %Miss	Part. L2 %Miss	Abs. %Imprv.	Rel. %Imprv.
art + mcf					
0.2	28.1	84.4	84.7	-0.3	-0.4
0.5		82.8	83.6	-0.8	-0.9
1		73.8	63.1	10.7	14.5
2		50.0	48.9	1.1	2.2
4		23.3	25.0	-1.7	-7.3
vpr + bzip2 + iu					
0.2	4.6	73.1	77.9	-0.8	-1.1
0.5		72.5	71.8	0.7	1.0
1		66.5	64.2	2.3	3.5
2		40.4	33.7	6.7	16.6
4		18.7	18.5	0.2	1.1
art1 + mcf1 + art2 + mcf2					
0.2	28.5	88.0	87.4	0.6	0.7
0.5		85.8	85.7	0.1	0.1
1		83.1	81.0	2.1	2.5
2		73.4	65.1	8.3	11.3
4		49.5	48.7	0.8	1.6

Table 4: Hit-rate Comparison between the standard LRU and the partitioned LRU.

relative to the miss-rate from the standard LRU replacement policy. For small caches, such as 256-KB and 512-KB caches, partitioning does not seem to help. We conjecture that the size of the total workloads is too large compared to the cache size. At the other extreme, partitioning cannot improve the cache performance if the cache is large enough to hold all the workloads.

The results demonstrate that on-line cache monitoring can be very useful for cache partitioning. Although the cache monitoring scheme is very simple and has a low implementation overhead, it can significantly improve the performance for some cases.

2.5 Analytical Models

Although the straightforward use of the marginal gain counters can improve performance, it is important to know its limitation. This section discusses analytical methods that can model the effects of memory contention amongst simultaneously-running processes, as well as the effects of time-sharing, using the information from the memory monitoring scheme. The model estimates the overall miss-rate when multiple processes execute simultaneously and concurrently. Estimating an overall miss-rate gives a better evaluation of a schedule or partition. First, a uniprocessor cache model for time-shared systems is briefly summarized. Then, the model is extended to include the effects of memory contention amongst simultaneously-running processes. Finally, a few examples of using the model with the monitoring scheme are shown.

2.5.1 Model for Time-Sharing

The time-sharing model from elsewhere [42] estimates the overall miss-rate for a fully-associative cache when multiple processes time-share the same cache (memory) on a uniprocessor system. There are three inputs to the model: (1) the memory size (C) in terms of the number of memory blocks (pages), (2) job sequences with the length of each process' time slice (T_i) in terms of the number of memory references, and (3) the miss-rate of each process as a function of cache space ($m_i(x)$). The model assumes that the least recently used (LRU) replacement policy is used, and there are no shared data structures among processes.

2.5.2 Extension to Space-Sharing

The original model assumes only one process executes at a time. In this subsection, we describe how the original model can be applied to multiprocessor systems where multiple processes can execute simultaneously sharing the memory (cache). We consider the situation where all processors context switch at the same time. More general cases where each processor can context switch at a different time can be modeled in a similar manner.

To model both time-sharing and space-sharing, we apply the original model twice. First, the model is applied to processes in the same time slice and generates a miss-rate curve for a time slice considering all processes in the time slice as one big process. Then, the estimated miss-rate curves are processed by the model again to incorporate the effects of time-sharing.

What should be the miss-rate curve for each time slice? Since the model for time-sharing needs *isolated* miss-rate curves, the miss-rate curve for each time-slice s is defined as the overall miss-rate of all processes in time slice s when they execute together without context switching using

memory of size x . We call this miss-rate curve for a time slice as a combined miss-rate curve $m_{combined,s}(x)$. Next we explain how to obtain the combined miss-rate curves.

The simultaneously executing processes within a time slice can be modeled as time-shared processes with very short time slices. Therefore, the original model is used to obtain the combined miss-rate curves by assuming the time slice is $ref_{s,p} / \sum_{i=1}^P ref_{s,i}$ for processor p in time-slice s . $ref_{s,p}$ is the number of memory accesses that processor p makes over time slice s .

Now we have the combined miss-rate curve for each time-slice. The overall miss-rate is estimated by using the original model assuming that only one process executes for a time slice whose miss-rate curve is $m_{combined,s}(x)$.

2.5.3 Model-Based Optimization

The analytical model can estimate the effects of both time-sharing and space-sharing using the information from our memory monitors. Therefore, our monitoring scheme with the model can be used for any optimization related to multi-tasking. For example, more accurate schedulers, which consider both time-sharing and space-sharing can be developed. Using the model, we can also partition the cache among concurrent processes or choose proper time quanta for them. In this subsection, we provide some preliminary examples of these applications.

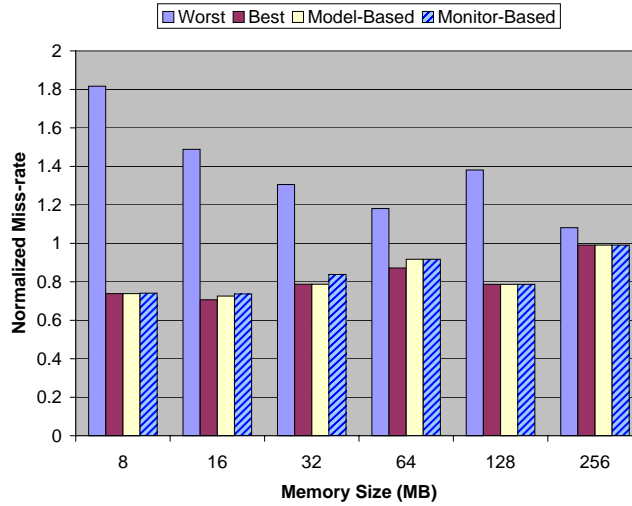


Figure 8: The comparison of miss-rates for various schedules: the worst case, the best case, the schedule based on the model, and the schedule decided by the algorithm in Section 2.3.

We applied the model to the same scheduling problem solved in Section 2.3. In this case, however, the model evaluates each schedule based on miss-rate curves from the monitor and decides the best schedule. Figure 8 illustrates the results. Although the improvement is small, the model-based scheduler finds better schedules than the monitor-based scheme for small memories.

The model is also applied to partition the cache space among concurrent processes. Some part of the cache is dedicated to each process and the rest is shared by all. Figure 9 shows the partitioning results when 8 processes (bzip2, gcc, swim, mesa, vortex, vpr, twolf, iu) are sharing the cache (32 KB, fully associative). The partition is updated every 10^5 cache references.

The figure demonstrates that time-sharing can degrade cache performance for some mid-range time quanta. Partitioning can eliminate the problem.

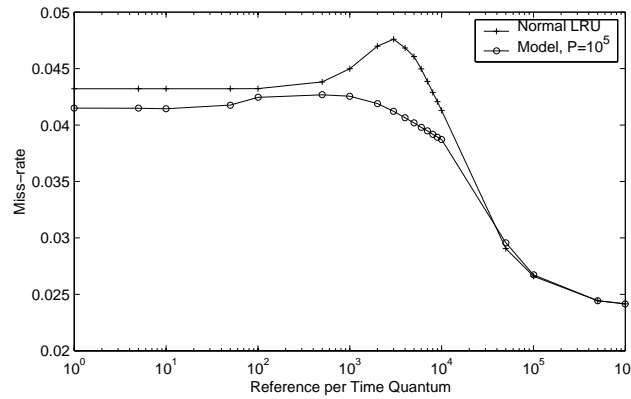


Figure 9: The results of cache partitioning among concurrent processes.

2.6 Example 2 – Dynamic Partitioning

The analytical cache model estimates the overall cache miss-rate for a multi-processing system. The cache size and the time quantum length for each job is known. The cache size is given by the number of cache blocks, and the time quantum is given by the number of memory references. Both are assumed to be constants (See Figure 10 (a)). In addition, associated with each job is its miss-rate curve, i.e., the number of cache misses as a function of the cache size.

This section explains the development of the model in several steps. Heavy use is made of the individual, isolated miss-rate curve (iimr). This curve is the miss-rate for a process as a function of cache size assuming no other processes are running. There is much information that can be gleaned from this equation. For example, we can compute the miss rate of a process as a function of time (Section 2.6.2) from the miss-rate of a process as a function of space.

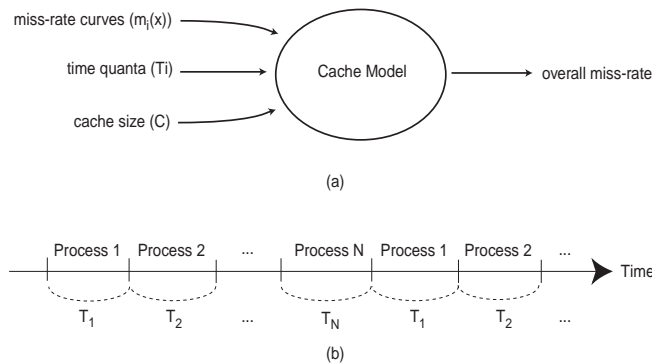


Figure 10: (a) The overview of an analytical cache model. (b) Round-robin schedule.

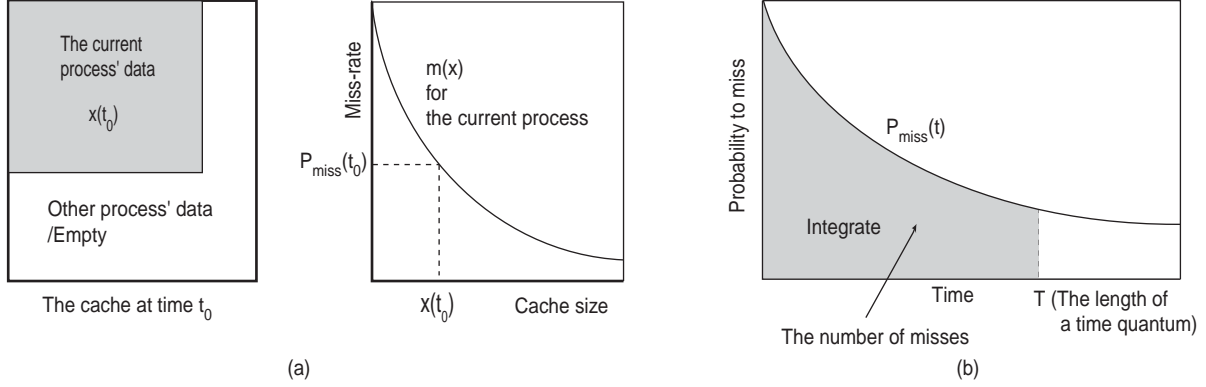


Figure 11: (a) The probability of a miss at time t_0 . (b) The number of misses from $P_{miss}(t)$ curve.

Observe that as a process executes, it either references an item in the cache, in which case its footprint size remains the same, or it gets a cache miss thereby increasing its footprint size. In other words, we know how much cache is allocated to a process as a function of time: from the iimr curve, we compute the independent, isolated footprint as a function of time (iifp) (Section 2.6.3).

If one knows how much cache is allocated to a process when it begins executing its time quantum and how much more cache it will need during the execution of that time quantum, we can compute how much cache will be left for the next process that is about to begin its time quantum execution. In other words, from the iifp curves of all the concurrent processes, we compute the individual, dependent footprint (dfp) as a function of time (Section 2.6.4).

At each time step, we know how much cache is allocated to the running process (from $dfp(t)$) and we know the miss rate for that size (from $iimr(S)$) for the executing process and so we can get the dependent miss rate as a function of time ($dmr(t)$) (Section 2.6.2).

Finally, integrating or summing the $dmr(t)$ over time, gives the overall average miss rate for a given cache size, given time quantum sizes, and a given set of concurrent processes (Section 2.6.5).

The following subsection gives an overview of our assumptions. The development of the cache model is then presented, following the outline given above. Finally, this section ends with experimental verification of the model.

2.6.1 Assumptions

The memory reference pattern of each process is assumed to be represented by a miss-rate curve that is a function of the cache size. Moreover, this miss-rate curve is assumed not to change over time. Although real applications do have dynamically changing memory reference patterns, our results show that, in practice, an average miss-rate function works very well. For abrupt changes in the reference pattern, multiple miss-rate curves can be used to estimate an overall miss-rate.

There is no shared address space among processes. This assumption is true for common cases where each process has its own virtual address space and the shared memory space is negligible compared to the entire memory space that is used by a process.

Finally, a round-robin scheduling policy with a fixed time quantum for each process is assumed

(see Figure 10 (b)), an LRU replacement policy is used, and the cache is fully associative. Although most real caches are set-associative, a model for fully-associative caches is very useful for understanding the effect of context switches because the model is simple. Moreover, cache partitioning experiments demonstrate that the fully-associative model can also be applied to set-associative caches in practice (Section 3). Elsewhere, we have extended the model to handle set-associative caches [19]. A model assuming many other scheduling methods and replacement policies can be similarly derived.

We make use of the following notations:

t the number of memory references from the beginning of a time quantum.

$x(t)$ the number of cache blocks that belong to a process after t memory references.

$m(x)$ the steady-state miss-rate for a process with cache size x .

T the number of memory references in a time quantum.

2.6.2 Cache Model

The goal is to predict the average miss-rate for a multiprocess machine with a given cache size and set of processes.

Given the independent, isolated miss-rate of a process as a function of cache size, we compute its miss-rate as a function of time. Let time t start at the beginning of a time quantum, not at the beginning of execution. Since all time quanta for a process are identical by our assumptions, we consider only one time quantum for each process.

Although the cache size is C , at certain times, it is possible that only part of the cache is filled with the current process' data (Figure 11 (a) shows a snapshot of a cache at time t_0). Therefore, the effective cache size at time t_0 can be thought of as the amount of the current process' data $x(t_0)$ in the cache at that time. The probability of a cache miss in the next memory reference is given by

$$P_{miss}(t_0) = m(x(t_0)). \quad (3)$$

Once we have $P_{miss}(t_0)$, it is easy to estimate the miss-rate over time during that time quantum. The number of misses for the process over a time quantum can be expressed as a simple integral, Figure 11 (b), where the miss-rate is expressed as the number of misses divided by the number of memory references.

$$\text{miss-rate} = \frac{1}{T} \int_0^T P_{miss}(t) dt = \frac{1}{T} \int_0^T m(x(t)) dt \quad (4)$$

2.6.3 Footprint as a function of time

We now estimate $x(t)$, the amount of a process' data, i.e. its footprint, in a cache as a function of time. Let us begin with the assumption that a process starts executing during a time quantum with an empty cache in order to estimate cache performance for cases when a cache gets flushed for every context switch. Virtual address caches without process ID are good examples of such a case. We show later how to estimate $x(t)$ when the cache is not empty at the start of a time quantum.

Consider $x^\infty(t)$ as the amount of the current process' data at time t for an infinite size cache. We assume that the process starts with an empty cache at time 0. There are two possibilities for $x^\infty(t)$ at time $t + 1$. If the $(t + 1)^{th}$ memory reference results in a cache miss, a new cache block is brought into the cache. As a result, the amount of the process's cache data increases by one block. Otherwise, the amount of data remains the same. Therefore, the amount of the process' data in the cache at time $t + 1$ is given by

$$x^\infty(t + 1) = \begin{cases} x^\infty(t) + 1 & (t + 1)^{th} \text{ reference misses} \\ x^\infty(t) & \text{otherwise.} \end{cases} \quad (5)$$

Since the probability for the $(t + 1)^{th}$ memory reference to miss is $m(x^\infty(t))$ from Equation 3, the expected value of $x(t + 1)$ can be written by

$$\begin{aligned} E[x^\infty(t + 1)] &= E[x^\infty(t) \cdot (1 - m(x^\infty(t))) \\ &\quad + (x^\infty(t) + 1) \cdot m(x^\infty(t))] \\ &= E[x^\infty(t) + 1 \cdot m(x^\infty(t))] \\ &= E[x^\infty(t)] + E[m(x^\infty(t))]. \end{aligned} \quad (6)$$

Assuming that $m(x)$ is convex⁵, we can use Jensen's inequality [3] and rewrite the equation as a function of $E[x^\infty(t)]$.

$$E[x^\infty(t + 1)] \geq E[x^\infty(t)] + m(E[x^\infty(t)]). \quad (7)$$

Usually, a miss-rate changes slowly. As a result, for a short interval such as from x to $x + 1$, $m(x)$ can be approximated as a straight line. Since the equality in Jensen's inequality holds if the function is a straight line, we can approximate the amount of data at time $t + 1$ as

$$E[x^\infty(t + 1)] \simeq E[x^\infty(t)] + m(E[x^\infty(t)]). \quad (8)$$

We can calculate the expectation of $x^\infty(t)$ more accurately by calculating the probability for every possible value at time t [19]. However, calculating a set of probabilities is computationally expensive. Also, our experiments show that the above approximation closely matches simulation results.

If we further approximate the amount of data $x^\infty(t)$ to be the expected value $E[x^\infty(t)]$, $x^\infty(t)$ can be expressed with a differential equation:

$$x^\infty(t + 1) - x^\infty(t) = m(x^\infty(t)), \quad (9)$$

which can be easily calculated in a recursive manner.

To obtain a closed form solution, we can rewrite the discrete form of the differential equation 9 to a continuous form:

$$\frac{dx^\infty}{dt} = m(x^\infty). \quad (10)$$

⁵If a replacement policy is smart enough, the marginal gain of having one more cache block monotonically decreases as we increase the cache size.

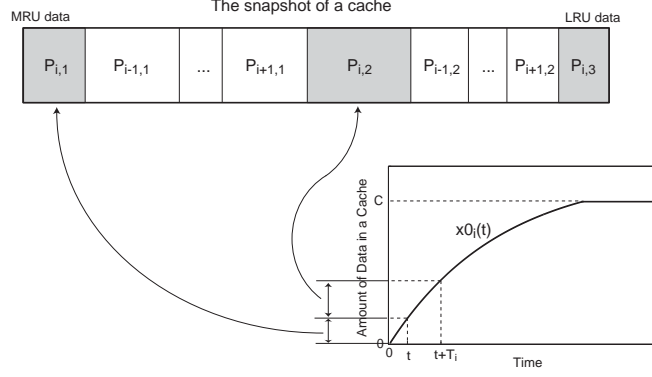


Figure 12: The snapshot of a cache after running Process i for time t .

Solving the differential equation by separating variables, the differential equation becomes

$$t = \int_{x^\infty(0)}^{x^\infty(t)} \frac{1}{m(x')} dx'. \quad (11)$$

We define a function $M(x)$ as an integral of $1/m(x)$, which means that $dM(x)/dx = 1/m(x)$, and then $x^\infty(t)$ can be written as a function of t :

$$x^\infty(t) = M^{-1}(t + M(x^\infty(0))) \quad (12)$$

where $M^{-1}(x)$ represents the inverse function of $M(x)$.

Finally, for a finite size cache, the amount of data in the cache is limited by the size of the cache C . Therefore, $x^\phi(t)$, the amount of a process' data starting from an empty cache, is written by

$$x^\phi(t) = \text{MIN}[x^\infty(t), C] = \text{MIN}[M^{-1}(t + M(0)), C]. \quad (13)$$

2.6.4 Individual, Dependent Footprint as a function of time

We now compute the amount of a process' data at time t when the cache is not flushed at a context switch, i.e., the dependent case. To distinguish between the processes, a subscript i is used to represent Process i . For example, $x_i(t)$ represents the amount of Process i 's data at time t .

The estimation of $x_i(t)$ is based on round-robin scheduling (See Figure 10 (b)) and the LRU replacement policy. Process i runs for a fixed length time quantum T_i . For simplicity, processes are assumed to be of infinite length so that there is no change in the scheduling. Also, the initial startup transient from an empty cache is ignored since it is negligible compared to the steady state.

To estimate the amount of a process' data at a given time, imagine the snapshot of a cache after executing Process i for time t as shown in Figure 12. Note that time is 0 at the beginning of the process' time quantum. In the figure, the blocks on the left side show recently used data, and blocks on the right side show old data. $P_{j,k}$ represents the data of Process j , and subscript k specifies the most recent time quantum when the data are referenced. From the figure, we can obtain $x_i(t)$ once we know the size of all $P_{j,k}$ blocks.

The size of each block can be estimated using the $x_i^\phi(t)$ curve from Equation 13, which is the amount of Process i 's data when the process starts with an empty cache. Since $x_i^\phi(t)$ can also be thought of as the amount of data that are referenced from time 0 to time t , $x_i^\phi(T_i)$ is the amount of data that are referenced over one time quantum. Similarly, we can estimate the amount of data that are referenced over k recent time quanta to be $x_i^\phi(k \cdot T_i)$. As a result, the size of Block $P_{j,k}$ can be written as

$$P_{j,k} = \begin{cases} x_j^\phi(t + (k-1) \cdot T_j) - x_j^\phi(t + (k-2) \cdot T_j) & \text{if } j \text{ is executing} \\ x_j^\phi(k \cdot T_j) - x_j^\phi((k-1) \cdot T_j) & \text{otherwise} \end{cases} \quad (14)$$

where we assume that $x_j^\phi(t) = 0$ if $t < 0$.

$x_i(t)$ is the sum of $P_{i,k}$ blocks that are inside the cache of size C in Figure 12. If we define $l_i(t)$ as the maximum integer value that satisfies the following inequality, then $l_i(t) + 1$ represents how many $P_{i,k}$ blocks are in the cache.

$$\sum_{k=1}^{l_i(t)} \sum_{j=1}^N P_{j,k} = x_i^\phi(t + (l_i(t) - 1) \cdot T_i) + \sum_{j=1, j \neq i}^N x_j^\phi(l_i(t) \cdot T_j) \leq C \quad (15)$$

where N is the number of processes. From $l_i(t)$ and Figure 12, the estimated value of $x_i(t)$ is

$$x_i(t) = \begin{cases} x_i^\phi(t + l_i(t) \cdot T_i) & \text{if } x_i^\phi(t + l_i(t) \cdot T_i) + \sum_{j=1, j \neq i}^N x_j^\phi(l_i(t) \cdot T_j) \leq C \\ C - \sum_{j=1, j \neq i}^N x_j^\phi(l_i(t) \cdot T_j) & \text{otherwise} \end{cases} \quad (16)$$

Figure 13 illustrates the relation between $x_i^\phi(t)$ and $x_i(t)$. In the figure $l_i(t)$ is assumed to be 2. Unlike the cache flushing case, a process can start with some of its data left in the cache. The amount of initial data $x_i(0)$ is given by Equation 16. If the least recently used (LRU) data in a cache does not belong to Process i , $x_i(t)$ increases the same as $x_i^\phi(t)$. However, if the LRU data belongs to Process i , $x_i(t)$ does not increase on a cache miss since Process i 's block gets replaced.

Define $t_{start}(j, k)$ as the time when the k^{th} MRU block of Process j ($P_{j,k}$) becomes the LRU part of a cache, and $t_{end}(j, k)$ as the time when $P_{j,k}$ gets completely replaced from the cache (See Figure 12). $t_{start}(j, k)$ and $t_{end}(j, k)$ specify the flat segments in Figure 13 and can be estimated from the following equations that are based on Equation 14.

$$x_j^\phi(t_{start}(j, k) + (k-1) \cdot T_j) + \sum_{p=1, p \neq j}^N x_p^\phi((k-1) \cdot T_p) = C. \quad (17)$$

$$x_j^\phi(t_{end}(j, k) + (k-2) \cdot T_j) + \sum_{p=1, p \neq j}^N x_p^\phi((k-1) \cdot T_p) = C. \quad (18)$$

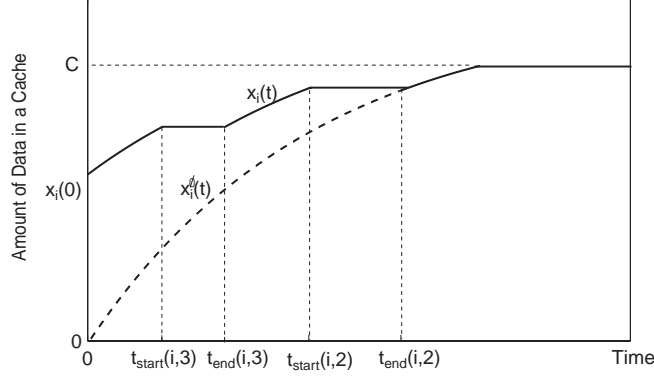


Figure 13: The relation between $x_i^\phi(t)$ and $x_i(t)$. $x_i(0)$ is the amount of Process i 's data in the cache when a time quantum starts.

$t_{start}(j, l_j(t) + 1)$ would be zero if Equation 17 is satisfied when $t_{start}(j, l_j(t) + 1)$ is negative, which means that the $P(j, l_j(t) + 1)$ block is already the LRU part of the cache at the beginning of a time quantum.

2.6.5 Overall Miss-rate

This section presents the overall miss-rate calculation. When a cache uses virtual address tags and gets flushed for every context switch, each process starts a time quantum with an empty cache. In this case, the miss-rate of a process can be estimated from the results of Section 2.6.2 and 2.6.3. From Equation 4 and 13, the miss-rate for Process i can be written by

$$\text{miss-rate}_i^\phi = \frac{1}{T_i} \int_0^{T_i} m_i(\text{MIN}[M_i^{-1}(t + M_i(0)), C]) dt. \quad (19)$$

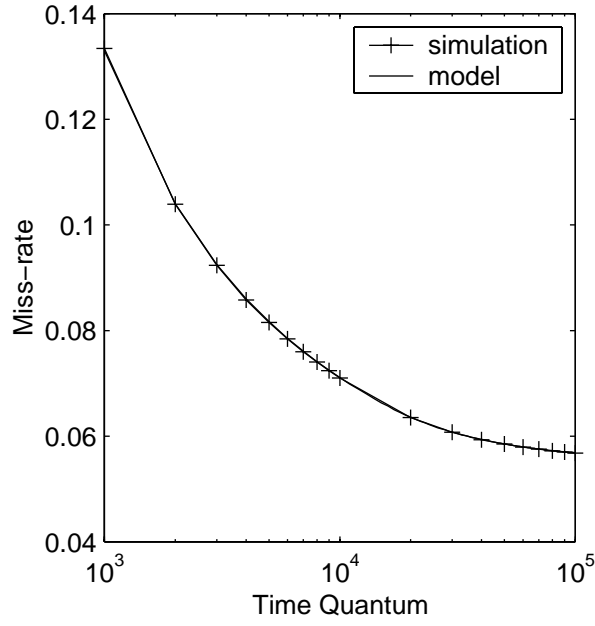
If a cache uses physical address tags or has a process' ID with virtual address tags, it does not have to be flushed at a context switch. In this case, the amount of data $x_i(t)$ is estimated in Section 2.6.4. The miss-rate for Process i can be written by

$$\text{miss-rate}_i = \frac{1}{T_i} \int_0^{T_i} m_i(x_i(t)) dt \quad (20)$$

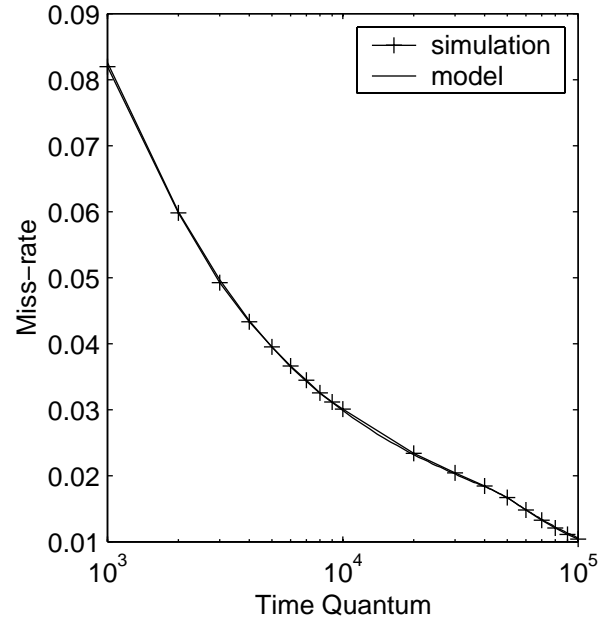
where $x_i(t)$ is given by Equation 16.

For actual calculation of the miss-rate, $t_{start}(j, k)$ and $t_{end}(j, k)$ from Equation 17 and 18 can be used. Since $t_{start}(j, k)$ and $t_{end}(j, k)$ specify the flat segments in Figure 13, the miss-rate of Process i can be rewritten by

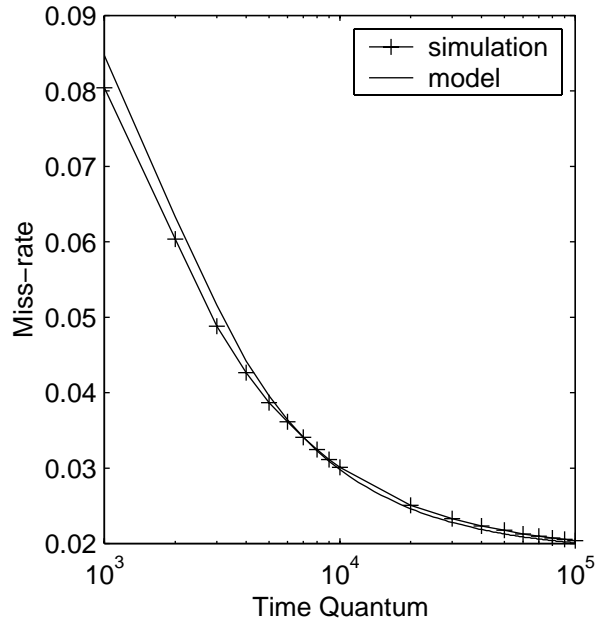
$$\begin{aligned} \text{miss-rate}_i = & \frac{1}{T_i} \left\{ \int_0^{T_i'} m_i(\text{MIN}[M_i^{-1}(t + M_i(x_i(0))), C]) dt \right. \\ & + \sum_{k=d_i}^{l_i(t)+1} m_i(x_i^\phi(t_{start}(i, k) + (k-1) \cdot T_i)) \\ & \cdot (\text{MIN}[t_{end}(i, k), T_i] - t_{start}(i, k)) \} \end{aligned} \quad (21)$$



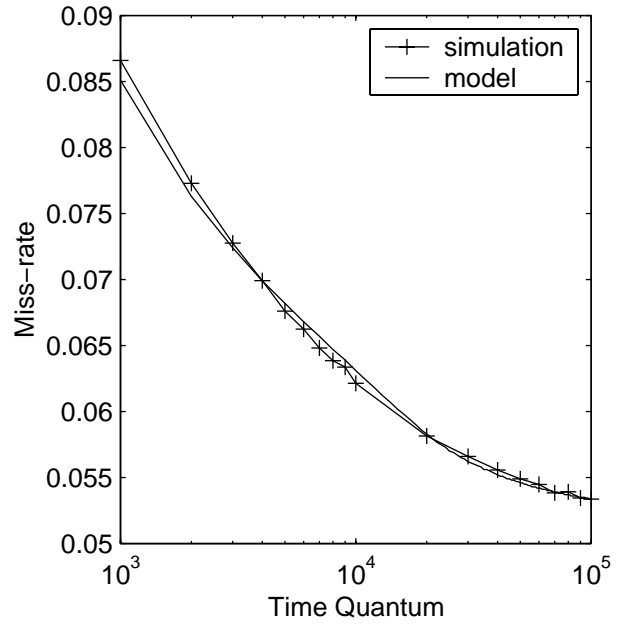
(a) vpr



(b) vortex



(c) gcc



(d) bzip2

Figure 14: The result of the cache model for cache flushing cases. (a) vpr. (b) vortex. (c) gcc. (d) bzip2.

where d_i is the minimum integer value that satisfies $t_{start}(i, d_i) < T_i$. T'_i is the time that Process i actually grows.

$$T'_i = T_i - \sum_{k=d_i}^{l_i(t)+1} (MIN[t_{end}(i, k), T_i] - t_{start}(i, k)). \quad (22)$$

As shown above, calculating a miss-rate could be complicated if we do not flush a cache at a context switch. If we assume that the executing process' data left in a cache is all in the most recently used part of the cache, we can use the equation for estimating the amount of data starting with an empty cache. Therefore, the calculation can be much simplified as follows,

$$\overline{\text{miss-rate}}_i = \frac{1}{T_i} \int_0^{T_i} m_i(MIN[M_i^{-1}(t + M_i(x_i(0))), C])dt \quad (23)$$

where $x_i(0)$ is estimated from Equation 16. The effect of this approximation is evaluated in the experiment section (cf. Section 2.7).

Once we calculate the miss-rate of each process, the overall miss-rate is straightforwardly calculated from those miss-rates.

$$\text{Overall miss-rate} = \frac{\sum_{i=1}^N \text{miss-rate}_i \cdot T_i}{\sum_{i=1}^N T_i} \quad (24)$$

2.7 Experimental Verification

Our cache model can be validated by comparing estimated miss-rate predictions with simulation results. Several combinations of benchmarks are modeled and simulated for various time quanta. First, we simulate cases when a cache gets flushed at every context switch, and compare the results with the model's estimation. Cases without cache flushing are also tested. For the cases without cache flushing, both the complete model (Equation 21) and the approximation (Equation 23) are used to estimate the overall miss-rate. Based on the simulation results, the error caused by the approximation is discussed.

2.7.1 Cache Flushing Case

The results of the cache model and simulations are shown in Figure 14 in cases when a process starts its time quantum with an empty cache. Four benchmarks from SPEC CPU2000 [32], which are `vpr`, `vortex`, `gcc` and `bzip2`, are tested. The cache is a 32-KB fully-associative cache with 32-Byte blocks. The miss-rate of a process is plotted as a function of the length of a time quantum, and shows a good agreement between the model's estimation and the simulation result.

As inputs to the cache model, the average miss-rate of each process has been obtained from simulations. Each process has been simulated for 25 million memory references, and the miss-rates of the process for various cache size have been recorded. The simulation results were also obtained by simulating benchmarks for 25 million memory references with flushing a cache every T memory references. As the result shows, the average miss-rate works very well.

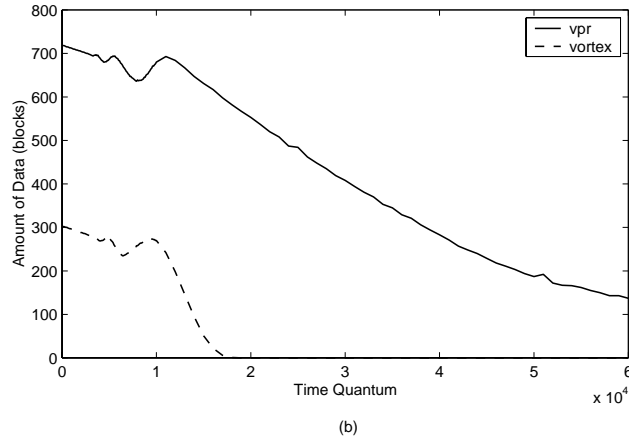
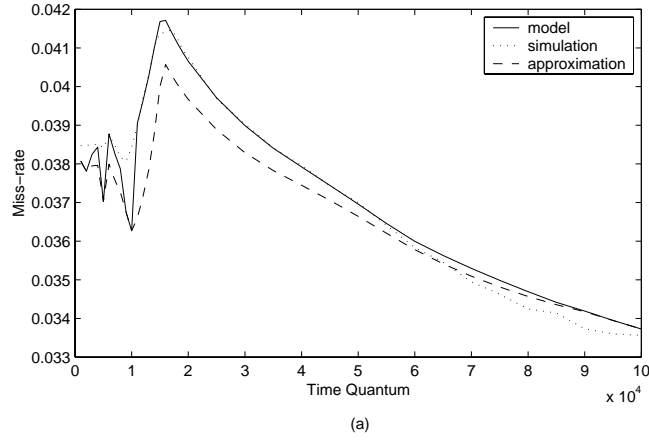


Figure 15: The result of the cache model when two processes (`vpr`, `vortex`) are sharing a cache (32 KB fully-associative). (a) the overall miss-rate. (b) the initial amount of data $x_i(0)$.

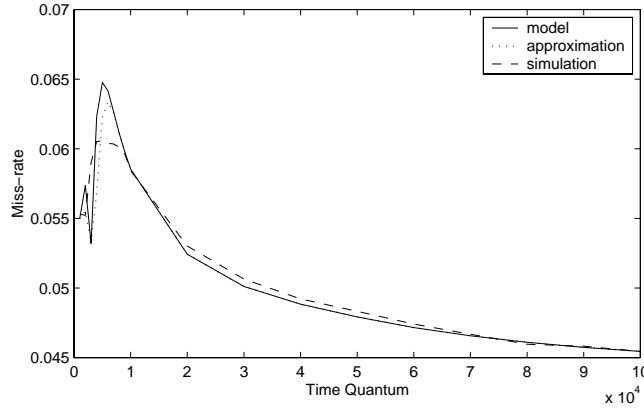


Figure 16: The overall miss-rate when four processes (*vpr*, *vortex*, *gcc*, *bzip2*) are sharing a cache (32 KB, fully-associative).

2.7.2 General Case

Figure 15 shows the result of the cache model when two processes are sharing a cache. The two benchmarks are *vpr* and *vortex* from SPEC CPU2000, and the cache is a 32-KB fully-associative cache with 32-Byte blocks. The overall miss-rates are shown in Figure 15 (a). As shown in the figure, the miss-rate estimated by the model shows a good agreement with the results of the simulations.

The figure also shows an interesting fact that a certain range of time quanta could be very problematic for cache performance. For short time quanta, the overall miss-rate is relatively small. For very long time quanta, context switches do not matter since a process spends most of its time in the steady state. However, medium time quanta could severely degrade cache miss-rates as shown in the figure. This problem occurs when a time quantum is long enough to pollute the cache but not long enough to compensate for the misses caused by context switches. The problem becomes clear in Figure 15 (b). The figure shows the amount of data left in the cache at the beginning of a time quantum. Comparing Figure 15 (a) and (b), we can see that the problem occurs when the initial amount of data rapidly decreases.

The error caused by our approximation (Equation 23) method can be seen in Figure 15. In the approximation, we assume that the data left in the cache at the beginning of a time quantum are all in the MRU region of the cache. In reality, however, the data left in the cache could be the LRU cache blocks and get replaced before other process' blocks in the cache, although the current process's data are likely to be accessed in the time quantum. As a result, the approximated miss-rate is lower than the simulation result when the initial amount of data is not zero.

A four-process case is also tested in Figure 16. Two more benchmarks, *gcc* and *bzip2*, from SPEC CPU2000 [32] are added to *vpr* and *vortex*, and the same cache configuration is used as the two process case. The figure also shows a very close agreement between the miss-rate estimated by the cache model and the miss-rate from simulations. The problematic time quanta and the effect of the approximation have changed. Since there are more processes polluting the cache as compared to the two process case, a process experiences an empty cache in shorter time

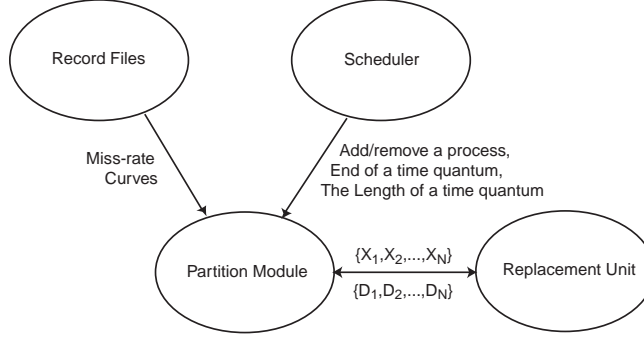


Figure 17: The implementation of on-line cache partitioning.

quanta. As a result, the problematic time quanta become shorter. On the other hand, the effect of the approximation is less harmful in this case. This is because the error in one process' miss-rate becomes less important as we have more processes.

3 Cache Partitioning

This section shows how the analytical cache model can be used to dynamically partition the cache. A partitioned cache allocates cache space to particular processes. This space is dedicated to the process and cannot be used to satisfy cache misses by other processes. Using trace-driven simulations, we compare partitioning with the normal LRU. The partitioning is based on the fully-associative cache model. However, simulation results demonstrate that this implementation works for both fully-associative caches and set-associative caches.

3.1 Recording Memory Reference Patterns

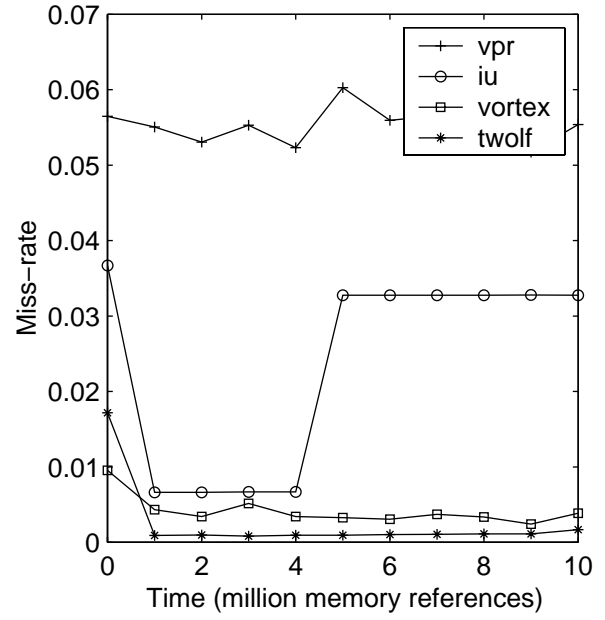
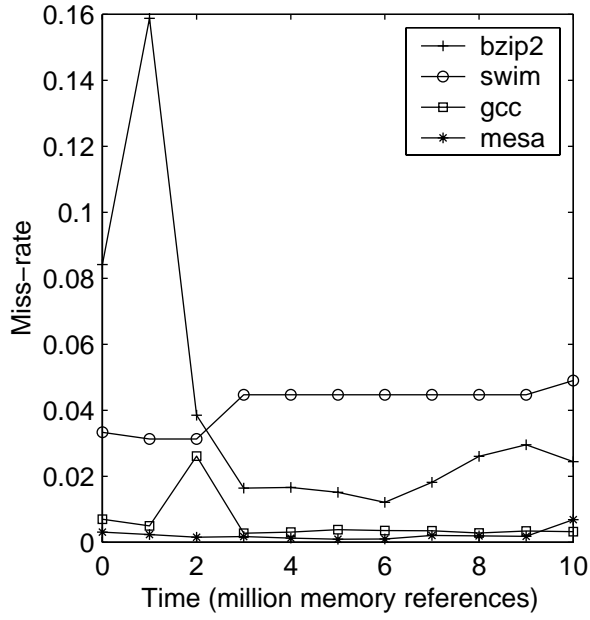
The miss-rate curves for each process are generated off-line. We record the miss-rate curve for each process to represent its memory reference pattern. For various cache sizes, a single process cache simulator is applied to each process. This information can be reused for any combination of processes as long as the cache configuration is the same⁶.

To incorporate the dynamically changing behavior of a process, a set of miss-rate curves, one for each time period, are produced. At run-time, the miss-rate curve is mapped to the appropriate time quantum.

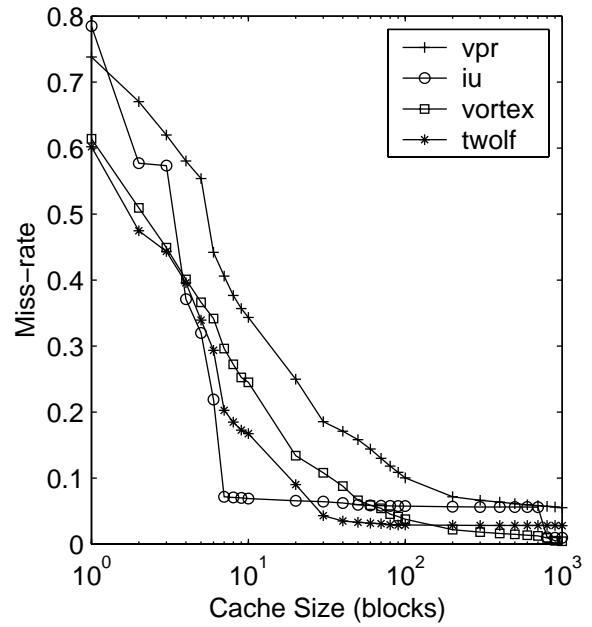
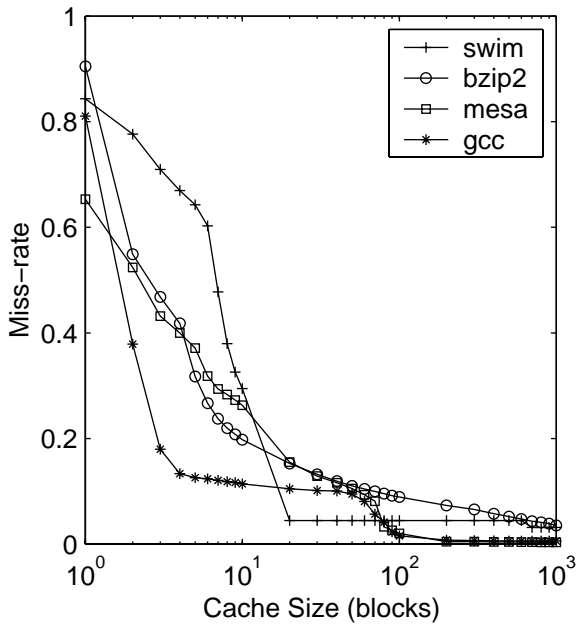
3.2 The Partitioning Scheme

The overall flow of the partitioning scheme can be viewed as a set of four modules: off-line recording, scheduler information, allocation, and replacement (Figure 17). The scheduler provides the partition module with the set of executing processes and their start/end times. The partition module

⁶Note that for our fully-associative model, only the cache block size matters



(a)



(b)

Figure 18: The characteristics of the benchmarks. (a) The change of a miss-rate over time. (b) The miss-rate as a function of the cache size.

uses the miss-rate information for the processes to calculate cache partitions at the end of each time quantum. Finally, the replacement unit maps these partitions to the appropriate parts of the cache.

The partition module decides the number of cache blocks that should be dedicated to a process (D_i). The D_i most recently used cache blocks of Process i are kept in the cache over other process' time quanta, and Process i starts its time quantum with those cache blocks in the cache. During its own time quantum, Process i can use all cache blocks that are not reserved for other processes ($S = C - \sum_{j=1, j \neq i}^N D_j$).

In addition to LRU information, our replacement decision depends on the number of cache blocks that currently belong to each process (X_i), that is, the number of cache lines in the cache that currently contain memory of that process. The LRU cache block of an active process (i) is chosen if its actual allocation (X_i) is larger than or equal to the desired one ($D_i + S \leq X_i$). Otherwise, the LRU cache block of a dormant overallocated process is chosen. For set-associative caches, there may be no cache block of the desired process in the set. In this case, the LRU cache block of the set is replaced.

For set-associative caches, the fully-associative replacement policy may result in replacing recently used data to keep useless data. Imagine the case when a process starts to heavily access two or more addresses that happen to be mapped to the same set. If the process already has many cache blocks in other sets, our partitioning will allocate only a few cache blocks in the accessed set for the process, causing lots of conflict misses. To solve this problem, we can use better mapping functions [22, 6] or a victim cache [8].

When a Process i first starts, D_i is set to zero since there is no cache block that belongs to the process. At the end of Process i 's time quantum, the partition module updates the information such as the miss-rate curve ($m_i(x)$) and the time quantum (T_i). If there is any change, D_i is also updated based on the cache model.

A cache partition specifies the amount of data in the cache at the beginning of a process' time quantum (D_i), and the maximum cache space the process can use ($C - \sum_{j=1, j \neq i}^N D_j$). Therefore, the number of misses for a process over one time quantum can be estimated from Equation 23:

$$\text{miss}_i = \int_0^{T_i} m_i(\text{MIN}[M_i^{-1}(t + M_i(D_i)), C - \sum_{j=1, j \neq i}^N D_j]) dt \quad (25)$$

where C is cache size, and N is the number of processes sharing the cache.

The new value of D_i is the integer, in the range $[0, X_i]$, that minimizes the total number of misses that is given by the following quantity:

$$\sum_{p=1}^N \int_0^{T_p} m_p(\text{MIN}[M_p^{-1}(t + M_p(D_p)), C - \sum_{q=1, q \neq p}^N D_q]) dt. \quad (26)$$

3.3 Experimental Verification

The case of eight processes sharing a 32-KB cache is simulated to evaluate model-based partitioning. Seven benchmarks (bzip2, gcc, swim, mesa, vortex, vpr, twolf) are from SPEC CPU2000 [32], and one (the image understanding program (iu)) is from a data intensive systems benchmark suite [38]. The overall miss-rate with partitioning is compared to the miss-rate only using the normal LRU replacement policy.

The simulations are carried out for fifty million memory references for each time quantum. Processes are scheduled in a round-robin fashion with the fixed number of memory references per time quantum. Also, the number of memory references per time quantum is assumed to be the same for the all eight processes. Finally, two record cycles (P), of ten million and one hundred thousand memory references, respectively, are used for the model-based partitioning. The record cycle represents how often the miss-rate curve is recorded for the off-line profiling. Therefore, a shorter record cycle implies more detailed information about a process' memory reference pattern.

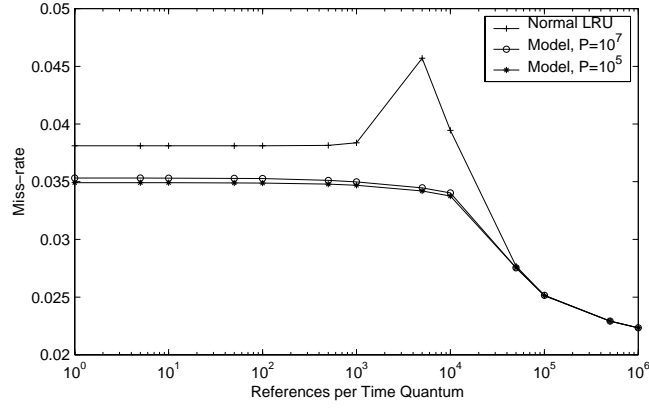
The characteristics of the benchmarks are illustrated in Figure 18. Figure 18 (a) shows the change of a miss-rate over time. The x-axis represents simulation time. The y-axis represents the average miss-rate over one million memory references at a given time. As shown in the figure, `bzip2`, `gcc`, `swim` and `iu` show abrupt changes in their miss-rate, whereas other benchmarks have very uniform miss-rate characteristics over time. Figure 18 (b) illustrates the miss-rate as a function of the cache size. For a 32-KB fully-associative cache, benchmarks show miss-rates between 1% and 5%.

3.3.1 Fully-Associative Result

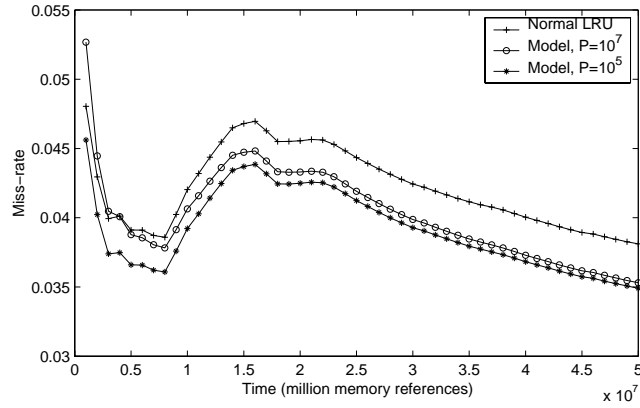
The results of cache partitioning for a fully-associative cache are shown in Figure 19. In Figure 19 (a), the miss-rates are averaged over 50 million memory references and shown for various time quanta. As discussed in the cache model, the normal LRU replacement policy is problematic for a certain range of time quanta. In this case, the overall miss-rate increases dramatically for time quanta between one thousand and ten thousand memory references. For this problematic region, the model-based partitioning improves the cache miss-rate by lowering it from 4.6% to 3.4%, which is about a 25% improvement. For short time quanta, the relative improvement is about 7%. For very long time quanta, the model-based partitioning shows the exact same result as the normal LRU replacement policy. In general, it is shown by the figure that the model-based partitioning always performs at least as well as or better than the normal LRU replacement policy. Also, the partitioning with a short record cycle performs better than the partitioning with a long record cycle.

In our example of a 32-KB cache with eight processes (Figure 19), the problematic time quanta are in the order of a thousand memory references, which is very short for modern microprocessors. As a result, only systems with very fast context switching, such as simultaneous multi-threading machines [47, 36, 30], can be improved for this cache size and workload. However, longer time quanta become problematic if a cache is larger. Therefore, conventional time-shared systems with very high clock frequency can also be improved by the same technique if a cache is large.

Figure 19 (b) shows the change of a miss-rate over time rather than an average miss-rate over the entire simulation. It is clear from the figure how the short record cycle helps partitioning. In the figure, the model-based partitioning with the long record cycle ($P = 10^7$) performs worse than LRU at the beginning of a simulation, even though it outperforms the normal LRU replacement policy overall. This is because the model-based partitioning has only one average miss-rate curve for a process. As shown in Figure 18, some benchmarks such as `bzip2` and `gcc` have a very different miss-rate at the beginning. Therefore, the average miss-rate curves for those benchmarks do not work at the beginning of the simulation, which results in worse performance than the normal LRU replacement policy. The model-based partitioning with the short record cycle ($P = 10^5$), on the other hand, always outperforms the normal LRU replacement policy. In this case, the model



(a)



(b)

Figure 19: The results of the model-based cache partitioning for a fully-associative cache when eight processes (bzip2, gcc, swim, mesa, vortex, vpr, twolf, iu) are sharing the cache (32 KB, fully associative). (a) the average miss-rate for various time quanta. (b) the change of the miss-rate over time with ten memory references per time quantum.

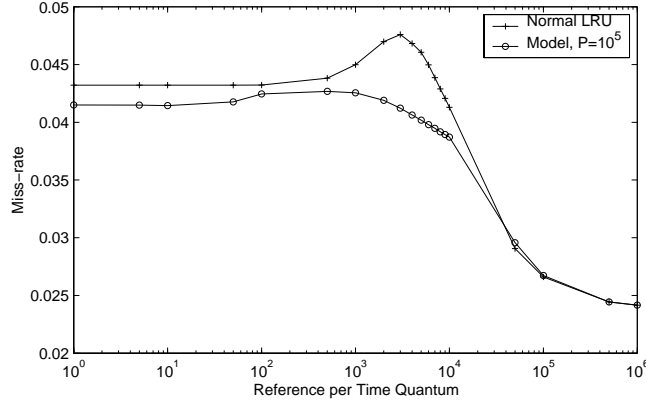


Figure 20: The results of the model-based cache partitioning for a set-associative cache when eight processes (bzip2, gcc, swim, mesa, vortex, vpr, twolf, iu) are sharing the cache (32 KB, 8-way associative).

has correct miss-rate curves for all the time quanta, and partitions the cache properly even for the beginning of processes.

3.3.2 Set-Associative Result

The result of cache partitioning for a set-associative cache is shown in Figure 20. The same set of benchmarks are simulated with a 32-KB 8-way set-associative cache, and the same miss-rate curves generated for a 32-KB fully-associative cache are used. In this case, a 16 entry victim cache is added. In the figure, the model-based partitioning improves the miss-rate about 4% for short time quanta and up to 15% for mid-range time quanta. The figure demonstrates that the model-based partitioning mechanism works reasonably well for set-associative caches.

4 Compressed Caches

One obvious way to increase the effective on-chip cache size is to use a dictionary-based compression scheme, however, since many values in the cache cannot be compressed naive implementations yield poor results. Our main innovation is to dynamically partition the cache in order to accommodate the various compression characteristics while keeping fast and simple access to the cache items.

Compression is a good match for caches since there is no assumption that a particular memory location will be found in the cache. Only performance suffers if an address is not found in the cache.

As main memory moves further away from the processor, it makes sense to spend a few extra cycles to avoid off-chip traversal costs. Our Partitioned Compressed Cache (PCC) algorithm is applied to the data values in the L2 cache where the decompress overheads are tolerable and the large size provides more opportunity for compression. PCC uses dictionary-based rather than sliding-window compression, to allow selective decompression of any cache line. PCC is described

and evaluated via three main issues: the compression scheme, the cache management scheme, and an evaluation mechanism.

The compression scheme of PCC is dictionary-based in which entries in the dictionary contain common strings of up to the size of a cache line. Cache lines can thus be compressed and decompressed on a line by line basis. PCC uses a “clock-scheme” that cycles over the compressed entries in the cache marking the corresponding dictionary entries as active while another “clock-scheme” cycles over the dictionary entries clearing out any inactive entries.

The cache entries can be compressed or not, and assuming that a cache line could be compressed down to $1/s$ of its uncompressed size, we allow each cache set to contain anywhere from n to sn entries. No matter what, sn address tags and LRU bits are maintained even though only n entries might be present in the set. When a new cache line is brought in or part of a cache line value is updated, PCC attempts to compress the line. A line is considered to be compressed if its compressed size is below a fixed threshold, e.g., a threshold of 16 bytes for a 32 byte cache line. Then, enough of the least recently used items are evicted from the cache so as to make room for the new item. In this way, the number of entries in a cache set varies depending on how compressible are the entries.

We use simulation but develop a new metric to evaluate performance. Reporting the number of compressible cache lines is not very helpful nor do cache hit ratios tell the whole story since they are very sensitive to the relative sizes of working set and the cache size. We use a metric that measures the effective cache size due to compression. That is, how much larger would a traditional cache need to be in order to achieve the same cache hit rates.

Of course the ultimate metric is performance, i.e., the reduction of the running time of the application. In particular, since we are targeting the memory system, the reduction in the average amount of time required to access data is of interest. For fairness, this comparison should take into account all area overheads and clock cycle penalties associated with compression. We have significant improvements in the time required to access data using a PCC.

After reviewing related work, the rest of the paper follows the three main issues: compression, cache management, and evaluation.

4.1 Related Work

While compression has been used in a variety of applications, it has yet to be researched extensively in the area of processor cache. Previous research includes compressing bus traffic to use narrower buses, compressing code for embedded systems to reduce memory requirements and power consumption, compressing file systems to save disk storage, and compressing virtual and main memory to reduce page faults.

Citron et al. [56] found an effective way to compact data and addresses to fit 32-bit values over a 16-bit bus, while Thumb [66, 61] and others [71, 64, 65] apply compression to instruction sets and binary executables. Douglis [58] proposed using a fast compression scheme [69] in a main memory system partition and shows several-fold speed improvement in some cases and substantial performance loss in others. Kjelson et al. [63] and Wilson et al. [70] consider an additional compressed level of memory hierarchy and found up to an order of magnitude speedup. IBM’s MXT technology [67] uses the scheme developed by Benveniste et al. [52] with 256 byte sub-blocks, a 1KB compression granularity, combining of partially filled blocks, along with the LZ77-like

parallel compression with shared dictionaries compression method.

Yang et al. [72, 73] explored compressing frequently occurring data values in focusing on direct-mapped L1 configurations and found that a large portion of cache data is made of only a few values, which they name Frequent Values. By storing data as small pointers to Frequent Values plus the remaining data, compression can be achieved. They propose a scheme where a cache line is compressed if at least half of its values are frequent values. They present results for direct-mapped L1 which with compression can become a 2-way associative cache with twice the capacity.

PCC is similar to Douglass's Compression Cache in its use of partitions to separate compressed and uncompressed data. A major difference is that Douglass's Compression Cache serves data to the higher level in the hierarchy only from the uncompressed partition, and so if the data requested is in the compressed partition, it is first moved to the uncompressed partition. The scheme developed by Benveniste et al. and the Frequent Value cache developed by Yang et al. serve data from both compressed and uncompressed representations as the PCC does, but both lack dynamic, adaptive partitioning.

4.2 The PCC Compression Algorithm

The dictionary compression scheme of PCC is based on the common Lempel-Ziv-Welch (LZW) compression technique [68]. When an entry is first placed in the cache or when an entry is modified, the dictionary is used to compress the cache line. The dictionary values are purged of useless entries by using a "clock-like" scheme over the compressed cache to mark all useful dictionary entries.

With LZW compression, the raw input stream data is compressed into another, shorter output stream of compressed symbols. Usually, the size of each uncompressed symbol, say of d bits, is smaller than the size of each compressed symbol, say of c bits. The dictionary initially consists of one entry for each uncompressed symbol. See Figure 21 for an example.

Compression works as follows. Find the longest prefix of the input stream that is in the dictionary and output the compressed symbol that corresponds to this dictionary entry. Extend the prefix string by the next input symbol and add it to the dictionary. The dictionary may either stop changing or it may be cleared of all entries when it becomes full. The prefix is removed from the input stream and the process continues.

Unlike LZW, PCC compresses only a cache line's worth of data at a time. A space-efficient dictionary representation maintains a table of 2^c entries, each of which contains two values: a compressed symbol that points to some other dictionary entry and an uncompressed symbol, for a total of $c + d$ bits per entry. The uncompressed symbols need not be explicitly stored in the dictionary as the first 2^d values represent themselves. Given a table entry, the corresponding string is the concatenation of the second value to the end of the string pointed to by the first value.

To compress a cache line, find the longest matching string, then output its dictionary symbol. Repeat until the entire line has been compressed. Decompression is much faster than compression. Each compressed symbol indexes into the dictionary to provide an uncompressed string. For a line containing n_c compressed symbols, $s_l/d - n_c$ table lookups are needed for decompression. The decompression latency can be improved by increasing the dictionary size and by parallelizing the table lookups. Naturally, increasing the compressed symbol size c while keeping the uncompressed symbol size d constant will increase the size of the associated table and enable more strings to be

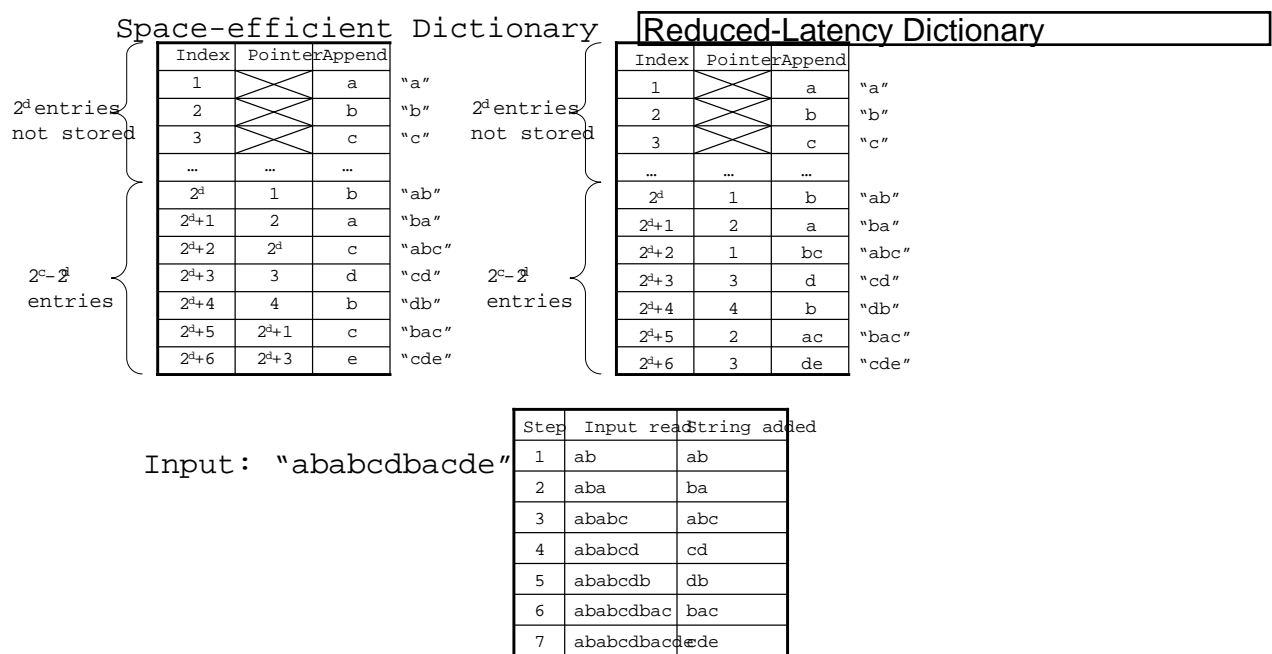


Figure 21: The space-efficient dictionary stores only one uncompressed symbol per entry, while the reduced-latency dictionary stores the entire string. The table at the lower half of the figure shows the order in which entries are added to the initially empty dictionaries.

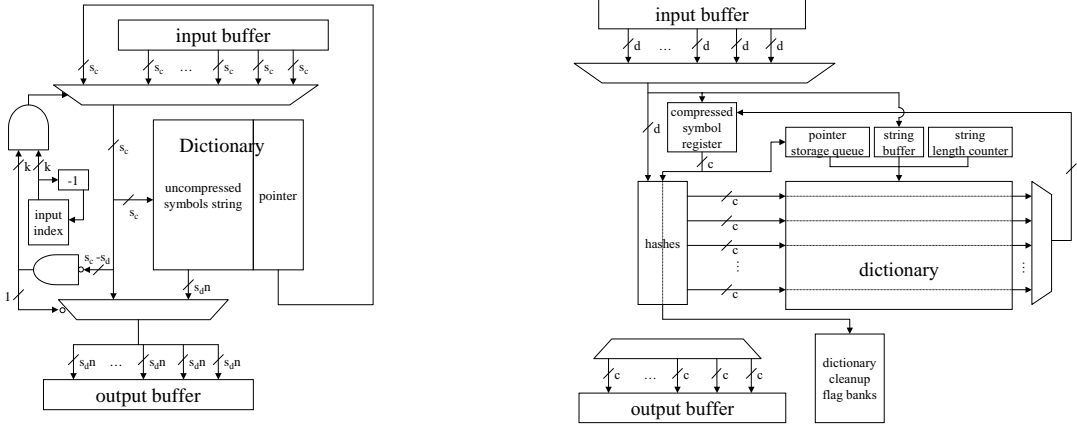


Figure 22: Compression and Decompression Logic: note that the design shown here does not include parallel compression or decompression, and therefore exhibits longer latencies for larger compression partition line sizes. n represents the number of uncompressed symbols stored in each dictionary entry.

stored.

One way to purge dictionary entries is by maintaining reference counts for each entry updated whenever a compressed cache line is evicted or replaced. PCC uses a more efficient method to purge entries sweeping through the contents of the cache slowly, using a clock scheme with two sets of flags. Each of the two sets has one flag per dictionary entry, and the status of the flag corresponds to whether or not the dictionary entry is used in the cache. If there is a flag in either set, it is assumed that the entry is being referenced. Compression or decompression also cause the appropriate dictionary entries flag to be set. A second process sweeps through the dictionary purging entries.

Compression time can be reduced dramatically by searching through only a strict subset of the entire dictionary for each uncompressed symbol of the input. A hash of the input is used to determine which entries to examine. If the dictionary is stored in multiple banks of memory, choosing hash functions such that entries are picked to be in separate banks allows these lookups to be done in parallel. Alternatively, content addressable memory (CAM) can be used to search all entries at the same time, reducing the number of dictionary accesses to the number of repetitions needed, or s_l/d accesses.

Decompression and compression logic is illustrated in Figure 22.

Decompression and compression can each be done in parallel to reduce their latency. To do so effectively, a method of performing multiple dictionary lookups in parallel is needed. One solution is to increase the number of ports to the dictionary. Another possibility is to keep several dictionaries, each with the same information. This provides a reduction in latency at the expense of the increased area needed for each additional dictionary. While decompressing, there are multiple compressed symbols which need to be decompressed. Since these symbols are independent of one another, they can be decompressed in parallel.

In practice, parallelizing the decompression process may not actually reduce latency signifi-

		LRU Entry		
		Compressed		Normal
New Entry	Compressed	Replace		convert to 2 entries
	Normal	If LRU ₂ compressed	If LRU ₂ normal	
		merge & replace	replace LRU ₂	Replace

Table 5: Replacement Algorithm

cantly. The experiments in this work show that performance is best when dictionary sizes are such that only one or two lookups are needed per compressed symbol. This is largely due to the low cost of increasing dictionary size in comparison to the benefits of decreasing the number of lookups.

4.3 PCC Management

Since not all data values are compressible nor are they all accessed at the same rate, an adaptive scheme can make the best use of the limited resources. An examination of the data values occurring in six benchmarks clearly show this variability, see Figure 23. For all unique cache lines accessed during the execution, the figure shows a histogram of the number of data values as a function of their compressibility. What is important to note is the variability between the benchmarks. Moreover, the overlaying curves show the usefulness of data; that is how often the data is referenced.

It is possible to statically partition the cache into compressed and uncompressed sections. Experiments reported elsewhere show that each benchmark requires a different partitioning to get performance improvements. Not only do data values differ in how much they can be compressed and how often they are accessed, cache sets have different access patterns. Some sets are “hot” experiencing a large number of accesses and some sets experience “thrashing” indicating they do not have enough capacity .

PCC uses the same basic storage allocation scheme of a traditional cache, i.e., a set of address tags and a set of data values. The number of address tags (and comparators) is fixed, but the number of cache lines in the set varies. For purposes of exposition, assume that two compressed entries require about the same number of bits as a single uncompressed entry; modifications to other ratios are straightforward but require more complicated circuitry. Let n be the number of normal entries, that is the data store for a set is $32n$ bytes. This space could store $2n$ compressed entries. In general $2i + j = 2n$ entries can be stored where j entries are compressed. Associated with each 32 byte field is one bit indicating whether there is one uncompressed or two compressed items present. The cache has $2n$ tags and maintains LRU information on all $2n$ entries, even though there may only be n entries actually in the cache.

On a cache hit, the cache line is either fetched directly from the data table or it must first be decompressed. Decompression is done via the dictionary as outlined in the previous section. Since compressed items enable a larger number of items to be stored in the cache, the overhead in

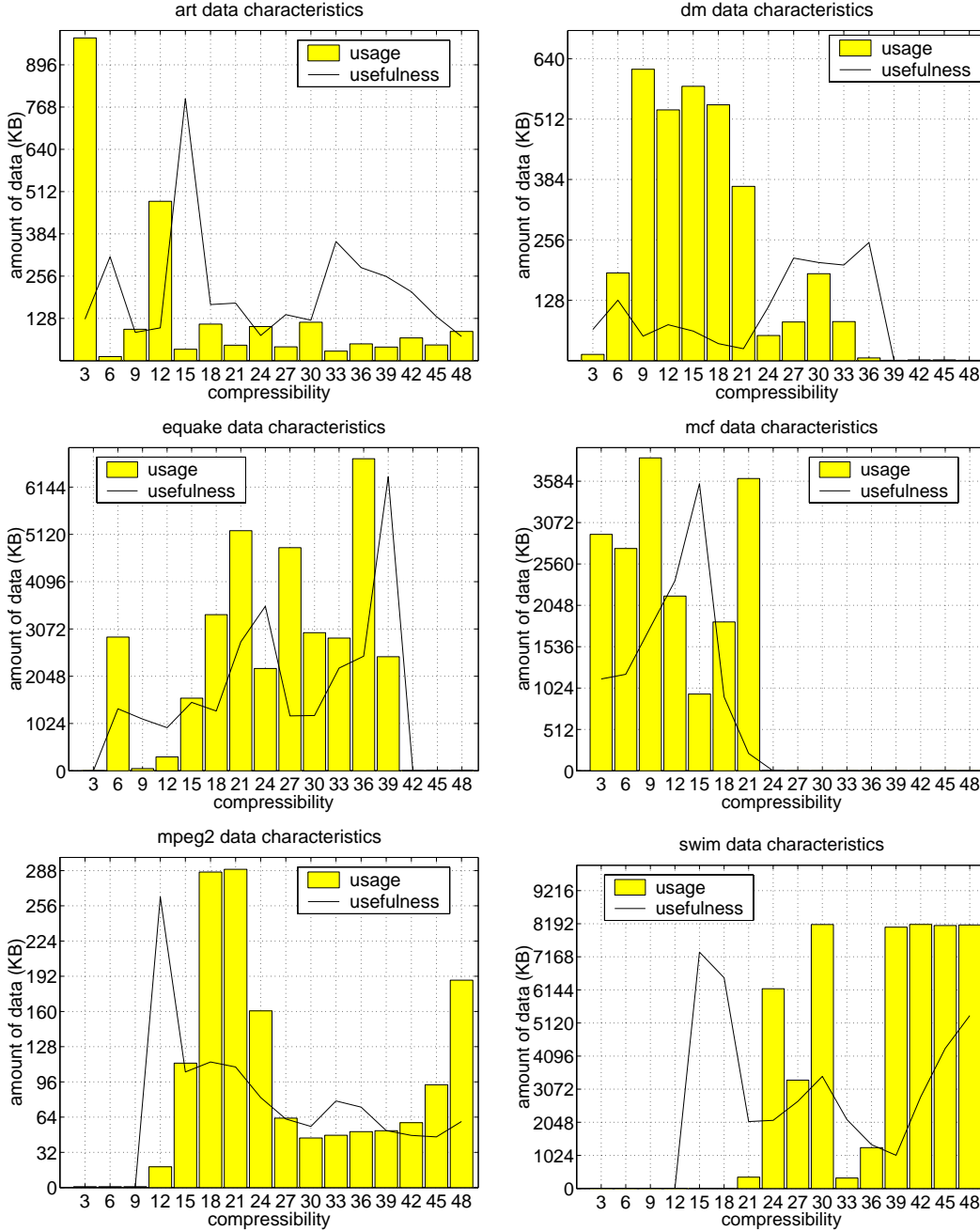


Figure 23: The histograms indicate the amount of data available at different levels of compressibility. The x-axis gives the size of the compressed line in bytes. The y-axis gives the amount of data in kilobytes, covering all unique memory addresses accessed during the simulation (an infinite sized cache). The top two histograms show that most data values are highly compressible, while the bottom-most right histogram shows that many data values would require more than 39 bytes to store a 32 byte cache line if compressed. The overlaying curves show the usefulness of data at different levels of compressibility. The y-axis gives the probability that a hit is on a particular cache line in the corresponding partition. This y value is equivalent to taking the total number of hits to the corresponding partition and dividing by the number of cache lines in that partition as given by the bars.

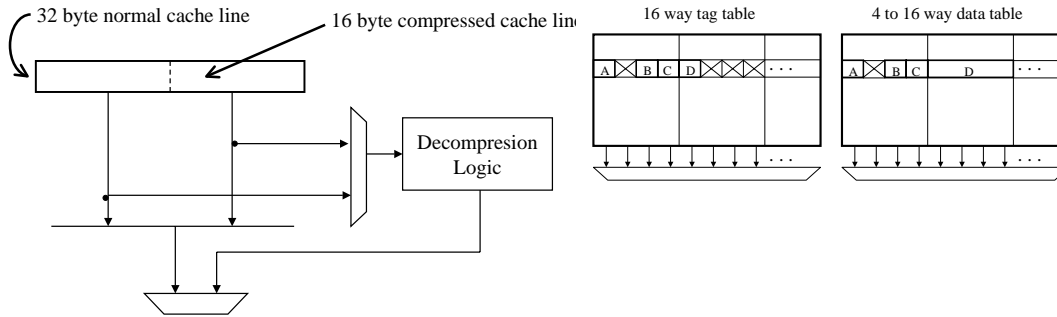


Figure 24: A sample configuration in which cache lines are assumed to be compressible to 1/2 their size, e.g. a 32 byte line compressed requires only 16 bytes. A cache set has tags and LRU bits for 16 ways, but as few as 8 or as many as 16 actually entries.

decompressing is offset by savings of not having to go off-chip to fetch the line from main memory.

4.3.1 The replacement strategy

The interesting situation happens during a cache miss, which we note in passing, is a rare event. When a cache miss occurs, the newly retrieved item will replace either the LRU item except sometimes it may replace two items. When the new cache line is fetched from main memory, an attempt is made to compress it. If the size of the compressed line is below a threshold, e.g., 16 bytes, then it is considered *compressible* otherwise it is *incompressible*. It must then be stored into the data table.

PPC Replacement Scheme: There are several cases to consider when inserting the cache line into the data table, see Table 5. The easiest case is when the new item and the LRU item have the same status – either both compressible or both incompressible – then the new item simply replaces the evicted LRU item. When the new item is compressible and the LRU item isn't, then the LRU item is evicted and its storage is converted to two compressible entries, one that is left empty (and marked as LRU). The case when the new item is incompressible has two subcases. If the two most LRU items are compressible, then they are both evicted and their space is converted to a normal entry. Finally, if the second most LRU item is incompressible, then it is the only one that is evicted and replaced; the LRU item remains in the cache.

The most work is required when the two least recently used items are both compressed and must be merged to form a normal entry. This might require moving an item if they do not form an even-odd pair. For example, if the compressed LRU₁ and LRU₂ are (LRU₁, X) and (LRU₂, Y), i.e., they are not an even-odd pair, if we wish to insert a new uncompressible block A into the cache, we will end up with A and (X, Y) in the set.

Note that this movement is only needed to be done during certain (not all) cache misses, a rare event, and could overlap the time to fetch the missed item. Also note that updating of the value of a part of a cache line can change its compressibility as well. Conceptually, updating can be considered to be an invalidation of the old entry and an insertion of a new entry. The above actions thus apply to the insertion.

In terms of cache hits, PCC always performs at least as well as a comparable standard cache. This is easy to see when one realizes that the LRU element in any set of a PCC cache was used no

more recently than the LRU element in the corresponding set of a standard cache. The PCC cache is a superset of the traditional cache.

4.3.2 A latency-sensitive replacement strategy

While the above replacement strategy works well in improving hit rates, it does not account for the fact that a cache hit to a compressed item has longer latency than a hit to an uncompressed item. It is possible to modify the strategy in a way that may slightly decrease the hit ratio while also significantly decreasing the number of times an item is decompressed.

In a level one (L1) cache, the most recently used item in a cache set usually experiences the most accesses. The second most recently used item in the set experiences the second most accesses, and so on. When the L1 cache is too small for the application, the level two (L2) cache behaves in a similar fashion. In other cases, accesses to the L2 cache are fairly random as to which element is accessed in a set. In such situations, it is helpful if the number of items in the set is large. In other words, the behavior of an L2 cache differs among applications and the behavior of each set in the L2 cache differs.

When the L2 behaves like an L1 cache, the MRU item will be frequently accessed. In such a case, it would be better if that item was stored in its uncompressed form. On the other hand, when items are accessed uniformly, it is better for the items to be compressed.

We modify the replacement scheme as follows.

Latency-sensitive Replacement Scheme: Whenever an MRU item is accessed and it is compressed, then it is replaced in the cache in its uncompressed form. This may cause the LRU item to be evicted. Whenever an LRU item is to be evicted from the cache and it is not compressed, PCC attempts to keep the item in the cache by compressing two other items in the set. The items of set are scanned from LRU to MRU order and for each item that is not compressed, an attempt is made to compress it. If two items are found, then they are compressed and the LRU item remains in the cache.

4.4 The Performance of PCC

We use simulation to evaluate the effectiveness of the PCC. Simulation is done using a hand-written cache simulator whose input consists of a trace of memory accesses. A trace of memory accesses is generated by the SimpleScalar simulator[53], which has been modified to dump a trace of memory accesses in a PDATS[60] formatted file. Applications are compiled with gcc or F90 with full optimization for the Alpha instruction set and then simulated with SimpleScalar. The benchmark applications are from the SPEC2000 benchmark suite and simulated for 30 to 50 million memory references.

The L1 cache is 16KB, 4 way set associative, with a 32 byte line size, and uses write-back. The L2 cache is simulated with varying size and associativity, with a 32 byte line size, and write-allocate (also known as fetch on write). We assume an uncompressed input symbol size d of 8 bits, and a compressed output symbol size c of 12 bits. The dictionary stores 16 uncompressed symbols per entry, making the size of the dictionary $(2^c - 2^d)(d * 16 + c)$, which evaluates to 537,600 bits, or 67,200 bytes.

4.4.1 Metrics

The common metric for the performance of a compression algorithm is to compare the sizes of the compressed and uncompressed data, i.e., the compression ratio [51], and for a cache is the miss rate reduction metric. However, the two configurations are not easily comparable as the partitioned cache uses more tags and comparators per area while at the same time using much less space to store data than the traditional cache.

We introduce the interpolated miss rate equivalent caches (IMRECs) metric that indicates the effective size of the cache. That is, how large must a standard cache be to have the same performance of a PCC cache. We wish to maximize the IMREC value; a value above 1 means that the PCC cache behaves like a larger-sized standard cache. For a given PCC configuration and miss rate, there is usually no naturally corresponding cache size with the same miss rate. Consequently, we interpolate linearly to calculate a fractional cache size. Our sample points are chosen by picking the size of a cache way, and then increasing the number of ways.

The size of a standard cache is the total number of cache lines multiplied by the cache line size. The size of a PCC cache includes the size of the dictionary, the additional address tag bits and additional status bits.

Let $M(C_j)$ be the miss rate of a j -way standard cache and $S(C)$ the size of cache C , the IMREC ratio is as follows:

$$\text{IMREC ratio} = S(C_i) + \frac{(S(C_{i+1}) - S(C_i))(M(C_i) - M(PCC))}{M(C_i) - M(C_{i+1})}$$

when $M(C_i) > M(PCC)$ and $M(C_{i+1}) < M(PCC)$

Another metric is the miss rate reduction (MRR), or the percent reduction in miss rate. But once again, we linearly interpolate to get the miss rate for an equivalently sized standard cache.

$$\text{Percent Miss Rate Reduction} = \left(MR(C_i) - \frac{(MR(C_i) - MR(C_{i+1}))(S(PCC) - S(C_i))}{S(C_{i+1}) - S(C_i)} \right) \times 100\%$$

when $S(C_i) \leq S(PCC)$ and $S(C_{i+1}) > S(PCC)$

It is important to understand what can cause large swings in IMREC ratio and MRR. Figure 25 shows typical curves of miss rate versus cache size. Miss rate curves typically have a prominent knee where miss rate decreases rapidly until the knee and then very slowly afterwards. The graph to the right shows that to the right of the knee, a small increase in MRR corresponds to a large increase in IMREC ratio. The graph to the left shows that to the left of the knee, a small increase in IMREC ratio corresponds to a large increase in MRR. While it may seem that the small miss rate improvements gained when to the right of the knee are unimportant, applications operating to the left of the knee are likely to be performing so badly that the issue of whether to use a PCC is not a primary concern. Thus most situations of interest occur to the right of the knee, where large IMREC ratios indicate that a PCC provides the same performance gains as a large cache but with much less hardware.

In our simulations, we account for the latency incurred by the decompression.

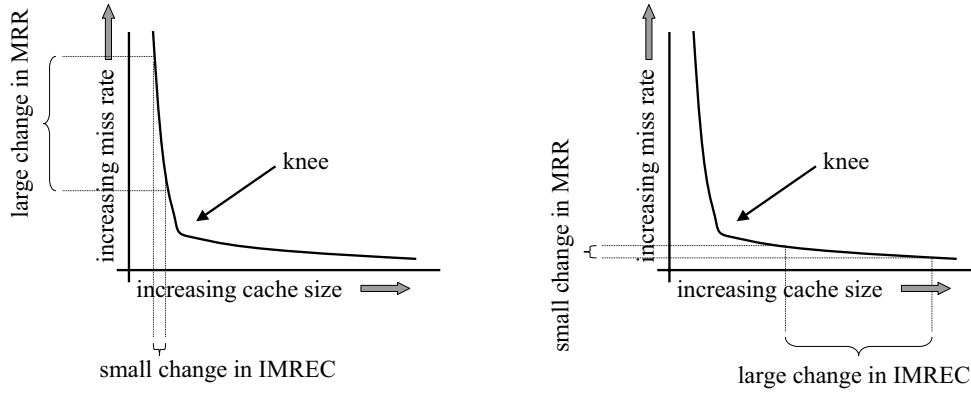


Figure 25: To the left of the knee, small increases in IMREC ratio correspond to large increases in MRR. To the right of the knee, small increases in MRR correspond to large increases in IMREC ratio.

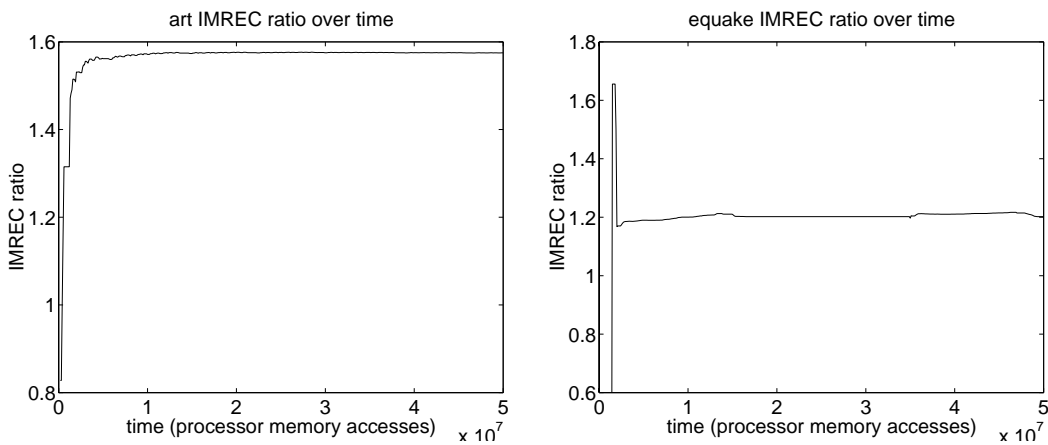


Figure 26: We plot the art and equake IMREC ratios over time so as to know how long to simulate before recording results.

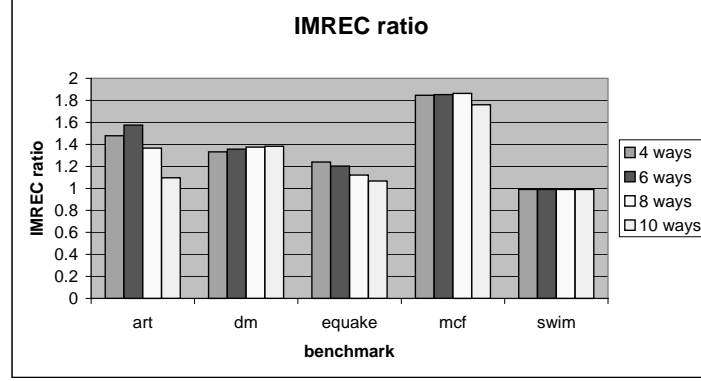


Figure 27: The IMREC values for five benchmarks show that PCC clearly improves the performance. PCC can sometimes perform like a cache 80 percent larger. It never performs worse than the same sized standard cache.

Let L_1 be the number of hits to L1, H_u be the number of hits in PCC to items that are not compressed, H_c be the number of hits to items that are compressed, and M_{pcc} be the number of L2 misses. Then, the time to access memory with a PCC is

$$MA_{pcc} = C_1 L_1 + C_2 H_u + C_3 H_c + C_4 M_{pcc}$$

For a standard cache, the number of L1 hits are the same, but the number of L2 hits and L2 misses differ. The time to access memory with a standard cache is:

$$MA_{sc} = C_1 L_1 + C_2 L_2 + C_4 M_{sc}$$

where C_1, C_2, C_3 , and C_4 are the times to access L1, L2, L2 compressed, and DRAM.

We define the *average L2 access time quotient* as,

$$ATQ_{PCC} = \frac{MA_{sc}}{MA_{PCC}}$$

Finally we must make sure that our simulations have run for long enough that the values presented are representative of the benchmark. We do this by plotting IMREC ratios over time. While some benchmarks like *mcf* clearly reach steady state quickly, others, like *equake*, have more varied behavior and take longer, as shown in Figure 26.

4.4.2 Results of the simulations

Although we ran many simulations over the large space of configurations, we present only one slice. Varying the dictionary size, the compressibility threshold (e.g. requiring a line to be 8 bytes or less before we consider it to be compressible), and many others result in too many graphs. We present what we believe to be a reasonable configuration that gives fairly good results.

The IMREC values for five benchmarks, Figure 27, show a performance improvement. PCC can sometimes perform like a cache 80 percent larger. It never performs worse than the same sized standard cache. Note that since IMREC takes into account the extra storage allocated to the dictionary, when PCC contains no compressed values, it will still be considered inferior to an equally sized cache.

When we take into account the latency to decompress a cache line, the results are less impressive, sometimes showing that PCC is slower than a standard cache, Figure 28.

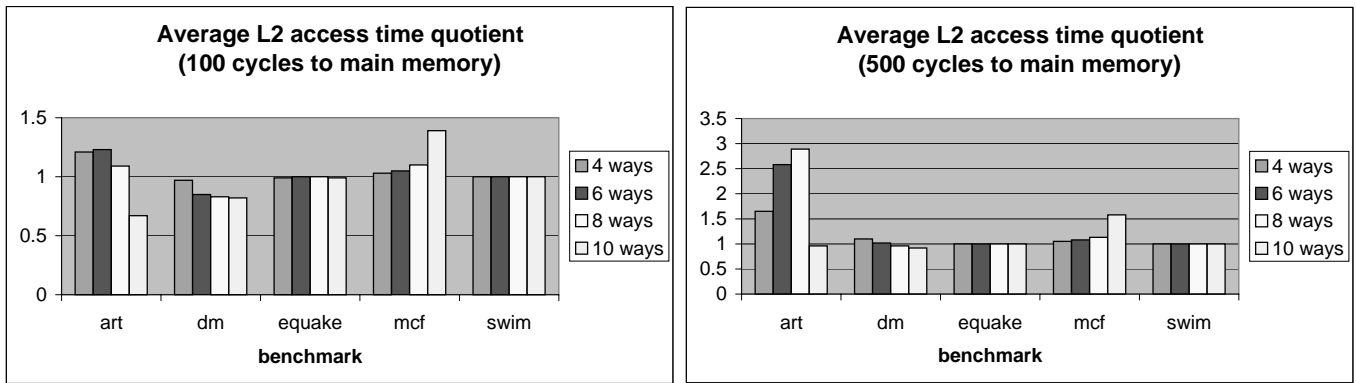


Figure 28: The ATQ values for five benchmarks show that even when the latency of decompressing cache items, PCC still can improve the performance although not for all applications. The graphs assume 100 or 500 cycle time to main memory.

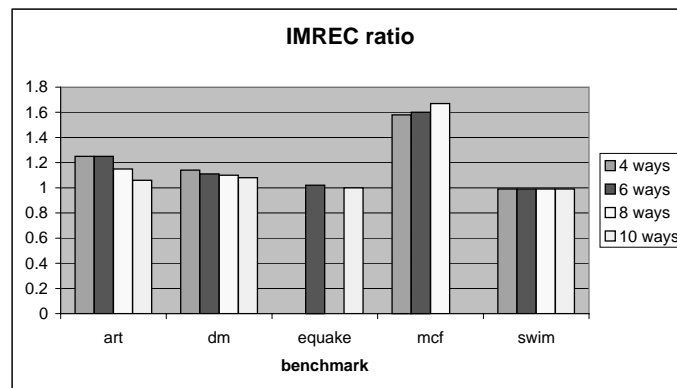


Figure 29: The IMREC values for the modified PCC replacement strategy that tries to keep the MRU item decompressed.

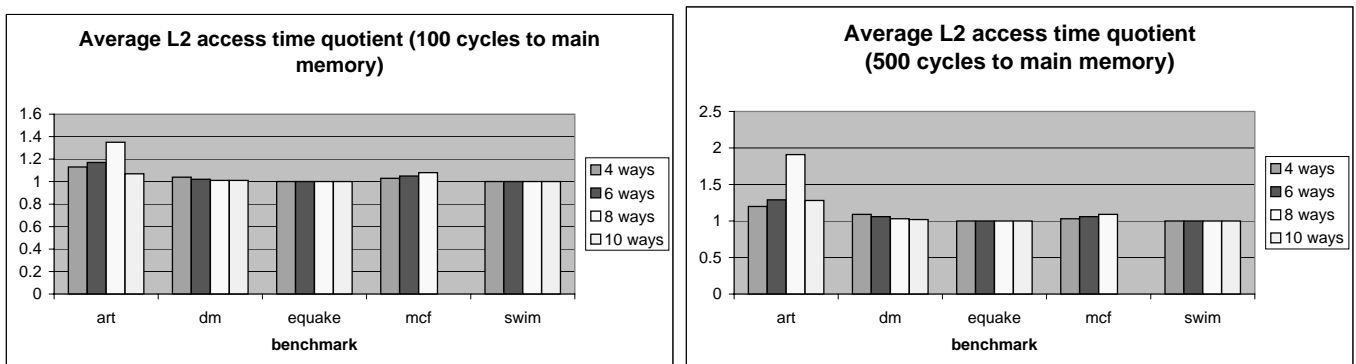


Figure 30: The ATQ values corresponding to Figure 29.

The latency-aware replacement scheme indeed reduces the overhead of decompressing without significantly reducing the number of hits. Figure 29 shows that the performance in terms of hits does not change very much, but the latencies numbers are much better, Note that access time quotients just compare the times and since the standard cache may access memory more frequently, PCC can do much better.

4.5 Conclusion

Compression can be added to caches to improve capacity, but creates problems of replacement strategy and fragmentation; these problems can be solved using partitioning. A dictionary-based compression scheme allows for reasonable compression and decompression latencies and compression ratios. Keeping the data in the dictionary from becoming stale can be avoided with a clock scheme.

The performance gains of a PCC over a standard cache of equivalent size can be attributed to two factors. A PCC potentially stores more data than a standard cache, which can reduce capacity misses. In addition, a PCC has more associativity than a standard cache of equivalent size, which can reduce conflict misses.

Various techniques can be used to reduce the latency involved in the compression and decompression process. Searching only part of the dictionary during compression, using multiple banks or CAMs to examine multiple dictionary entries simultaneously, and compressing a cache line starting at different points in parallel can reduce compression latency. Decompression latency can be reduced by storing more symbols per dictionary entry and decompressing multiple symbols in parallel. There are many different compression schemes some of which may perform better or be easier to implement in hardware.

The benefits of having a partitioned compressed cache have not yet been fully explored. For example, CRCs of the cache data can be done for only a small incremental cost, an idea which is proposed also in [67]. The partitioning based on compressibility may also naturally improve the performance of a processor running multiple jobs, some of which are streaming applications. The streaming data is likely to be hard to compress, and can therefore automatically be placed into its own partition separate from non-streaming data.

References

- [1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2), May 1989.
- [2] Compaq. Compaq alphastation family.
- [3] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, John & Sons, Incorporated, Mar. 1991.
- [4] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5), 1997.

- [5] C. Freeburn. The Hewlett Packard PA-RISC 8500 processor. Technical report, Hewlett Packard Laboratories, Oct. 1998.
- [6] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *the 1997 international conference on Supercomputing*, 1997.
- [7] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.
- [8] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *the 17th Annual International Symposium on Computer Architecture*, 1990.
- [9] D. B. Kirk. Process dependent static cache partitioning for real-time systems. In *Real-Time Systems Symposium*, 1988.
- [10] H. Kwak, B. Lee, A. R. Hurson, S.-H. Yoon, and W.-J. Hahn. Effects of multithreading on cache performance. *IEEE Transactions on Computers*, 48(2), Feb. 1999.
- [11] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15, 1997.
- [12] P. Magnusson and B. Werner. Efficient memory simulation in SimICS. In *28th Annual Simulation Symposium*, 1995.
- [13] MIPS Technologies, Inc. *MIPS R10000 Microprocessor User's Manual*, 1996.
- [14] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *the fourth international conference on Architectural support for programming languages and operating systems*, 1991.
- [15] J. Muoz. *Data-Intensive Systems Benchmark Suite Analysis and Specification*. <http://www.aaec.com/projectweb/dis>, June 1999.
- [16] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology*, 1995.
- [17] M. S. Squillante and E. D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2), Feb. 1993.
- [18] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9), Sept. 1992.
- [19] G. E. Suh and L. Rudolph. Set-associative cache models for time-shared systems. Technical Report CSG Memo 433, Massachusetts Institute of Technology, 2001.

- [20] D. Thiébaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4), Nov. 1987.
- [21] D. Thiébaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6), June 1992.
- [22] N. Topham and A. González. Randomized cache placement for eliminating conflicts. *IEEE Transactions on Computers*, 48(2), Feb. 1999.
- [23] J. Torrellas, A. Tucker, and A. Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors: A summary. In *the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1993.
- [24] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, 1995.
- [25] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2), June 1997.
- [26] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R1000. In *Supercomputing'96*, 1996.
- [27] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2), May 1989.
- [28] Compaq. Compaq AlphaServer series. <http://www.compaq.com>.
- [29] W. J. Dally, S. Keckler, N. Carter, A. Chang, M. Filo, and W. S. Lee. M-Machine architecture v1.0. Technical Report Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology, 1994.
- [30] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5), 1997.
- [31] D. G. Feitelson and L. Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Journal of Parallel and Distributed Computing*, 1996.
- [32] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.
- [33] HP. HP 9000 superdome specifications. <http://www.hp.com>.
- [34] IBM. RS/6000 enterprise server model S80. <http://www.ibm.com>.
- [35] Intel. *Intel StrongARM SA-1100 Microprocessor*, April 1999.
- [36] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15, 1997.

- [37] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *the fourth international conference on Architectural support for programming languages and operating systems*, 1991.
- [38] J. Munoz. *Data-Intensive Systems Benchmark Suite Analysis and Specification*. <http://www.aaec.com/projectweb/dis>, June 1999.
- [39] A. Snaveley and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.
- [40] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9), Sept. 1992.
- [41] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems*, 1995.
- [42] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with application to cache partitioning. In *the 15th international conference on Supercomputing*, 2001.
- [43] G. E. Suh, S. Devadas, and L. Rudolph. Dynamic cache partitioning for simultaneous multithreading systems. In *Thirteenth IASTED International Conference on Parallel and Distributed Computing System*, 2001.
- [44] G. E. Suh, L. Rudolph, and S. Devadas. Effects of memory performance on parallel job scheduling. In *7th International Workshop on Job Scheduling Strategies for Parallel Processing (in LNCS 2221)*, pages 116–132, 2001.
- [45] D. Thiébaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4), Nov. 1987.
- [46] D. Thiébaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6), June 1992.
- [47] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, 1995.
- [48] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R1000 performance counters. In *Supercomputing'96*, 1996.
- [49] M. Zhang and K. Asanović. Highly-associative caches for low-power processors. In *Kool Chips Workshop in 33rd International Symposium on Microarchitecture*, 2000.
- [50] Data Compression Conference.
- [51] B. Abali, H. Franke, X. Shen, D. Poff, and T. B. Smith. Performance of hardware compressed main memory, 2001.

- [52] C. Benveniste, P. Franaszek, and J. Robinson. Cache-memory interfaces in compressed memory systems. In *IEEE Transactions on Computers, Volume #50 number 11*, November 2001.
- [53] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. In *Technical report, University of Wisconsin-Madison Computer Science Department*, 1997.
- [54] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 2–9, October 1992.
- [55] D. T. Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching*. PhD thesis, Massachusetts Institute of Technology, 1999.
- [56] Daniel Citron and Larry Rudolph. Creating a wider bus using caching techniques. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 90–99, Raleigh, North Carolina, 1995.
- [57] M. Clark and S. Rago. The Desktop File System. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 113–124, Boston, Massachusetts, 6–10 1994.
- [58] Fred Douglass. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, San Diego, California, 1993.
- [59] Peter A. Franaszek, John T. Robinson, and Joy Thomas. Parallel compression with cooperative dictionary construction. In *Data Compression Conference*, pages 200–209, 1996.
- [60] E. E. Johnson and J. Ha. PDATS: Lossless addresss trace compression for reducing file size and access time. In *IEEE International Phoenix Conference on Computers and Communications*, 1994.
- [61] Kevin D. Kissell. MIPS16: High-density MIPS for the embedded market. In *Proceedings of Real Time Systems '97 (RTS97)*, 1997.
- [62] M. Kjelso, M. Gooch, and S. Jones. Design and performance of a main memory hardware data compressor. In *Proceedings of the 22nd Euromicro Conference*, pages 423–430, September 1996.
- [63] M. Kjelso, M. Gooch, and S. Jones. Performance evaluation of computer architectures with main memory data compression, 1999.
- [64] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 194–203, Research Triangle Park, North Carolina, December 1997.
- [65] S. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, Massachusetts Institute of Technology, June 1996.

- [66] Simon Segars, Keith Clarke, and Liam Goudge. Embedded control problems, Thumb, and the ARM7TDMI. *IEEE Micro*, 15(5):22–30, 1995.
- [67] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. Wazlowski, and P. M. Bland. IBM Memory Expansion Technology (MXT). In *IBM Journal of Research and Development* vol. 45, No. 2, pages 271–285, March 2001.
- [68] T. Welch. High speed data compression and decompression apparatus and method, US Patent 4,558,302, December 1985.
- [69] Ross N. Williams. An extremely fast Ziv-Lempel compression algorithm. In *Data Compression Conference*, pages 362–371, April 1991.
- [70] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of 1999 Summer USENIX Conference*, pages 101–116, Monterey, California, 1999.
- [71] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proceedings of the 25th International Symposium on Microarchitecture*, Portland, Oregon, December 1992.
- [72] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *33rd International Symposium on Microarchitecture*, Monterey, CA, December 2000.
- [73] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. In *The 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.